Design and development of a multi-platform software development kit of a mobile medical device

Daniel Eke

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 6.5.2019

Supervisor

Assoc. Prof. Casper Lassenius

Advisor

Teemu Piirainen

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Daniel Eke

Title Design and development of a multi-platform software development kit of a mobile medical device

Degree programme Master's Programme in ICT Innovation

Major Software and Service Architectures **Code of major** SCI3082

Supervisor Assoc. Prof. Casper Lassenius

Advisor Teemu Piirainen

Date 6.5.2019 **Number of pages** 76 **Language** English

Abstract

This thesis includes a literature study of best practices for development of mobile SDKs. These techniques are verified through a practical business case provided by a startup company in Finland. The main product of the company is a mobile medical device, which requires an SDK for Android and iOS devices. The practical development process presents the proof of concept implementation of the core parts of this SDK.

The selected technology for the implementation is Kotlin/Native, which allows multi-platform programming in Kotlin. This feature is still in an experimental stage as of Spring 2019. An additional goal of this thesis is to evaluate the production application of the selected technology.

In the conclusion, a list of best practices for a general SDK design and development process is presented. These are successfully applied during the implementation of the SDK, which meets the requirements of the business case provider company. At the time of this thesis work, multi-platform development in Kotlin/Native is still not the best option for an SDK implementation. However, the experimentation resulted in an open-source configuration example, which might be used in future projects.

Keywords SDK, API, Cross-Platform, Kotlin, Refactoring, Software Architecture, iOS, Android

Contents

Abstract 3

Contents 4

Abbreviations 6

1 Introduction 7
- 1.1 Background . . . 8
  - 1.1.1 Academic context . . . 8
  - 1.1.2 Industrial context . . . 8
- 1.2 Problem statement . . . 9
- 1.3 Structure of the Thesis . . . 10

2 Methodology 11
- 2.1 Best practice research . . . 11
- 2.2 Development . . . 12
- 2.3 Evaluation . . . 13

3 Best practices research 14
- 3.1 Literature study . . . 14
  - 3.1.1 Key concepts . . . 14
  - 3.1.2 SDK development principles . . . 16
- 3.2 Analysing public SDKs . . . 18
- 3.3 Development framework selection . . . 21

4 Software architecture design 23
- 4.1 Requirements specification . . . 23
- 4.2 Codebase analysis . . . 27
  - 4.2.1 High-level overview . . . 28
  - 4.2.2 Source code overview . . . 30
- 4.3 Architectural plan . . . 32
- 4.4 Summary . . . 35

5 Implementation 36
- 5.1 Kotlin/Native . . . 36
  - 5.1.1 Brief overview of Kotlin . . . 36
  - 5.1.2 The use-case of Kotlin/Native . . . 37
  - 5.1.3 Platform-specific Kotlin source . . . 37
- 5.2 Configuration . . . 39
  - 5.2.1 Bundling existing code . . . 39
  - 5.2.2 Dependencies . . . 41
  - 5.2.3 Open-source collaboration . . . 42
  - 5.2.4 Distribution . . . 44
  - 5.2.5 Summary . . . 45

5.3 API . . . 45
5.4 Refactoring . . . 48
5.4.1 Separation of layers . . . 48
5.4.2 Elevating the abstraction . . . 50
5.4.3 Device sync process management . . . 51
5.4.4 Persistence . . . 54
5.5 Distribution . . . 56
5.5.1 API documentation . . . 56
5.5.2 Sample applications . . . 59

6 Evaluation and discussion **60**
6.1 Meeting the requirements . . . 60
6.1.1 Functional requirements . . . 60
6.1.2 Interviews . . . 61
6.1.3 Security evaluation . . . 66
6.2 Development with Kotlin/Native . . . 67
6.2.1 Challenges . . . 68
6.2.2 Highlights . . . 68
6.3 Best practices evaluation . . . 69
6.4 Answering the research questions . . . 70

7 Summary **72**
7.1 Conclusion . . . 72
7.2 Further development . . . 73
7.3 Future of Kotlin/Native . . . 73

8 Bibliography **75**

Abbreviations

SDK	Software Development Kit
API	Application Programming Interface
UI	User Interface
GUI	Graphical User Interface
OS	Operating System
IoT	Internet of Things
IoC	Inversion of Control
UML	Unified Modeling Language
JVM	Java Virtual Machine
LLVM	Low Level Virtual Machine
IDE	Integrated Development Environment
CPU	Central Processing Unit
AAR	Android Archive Library
BLE	Bluetooth Low Energy
MASVS	Mobile Application Security Verification Standard
ABI	Application Binary Interface

1 Introduction

Since 2007, when Apple released the first iPhone, the smartphone market has changed the way how we interact with computers. During the last ten years, many new IoT devices have been released, which are directly connected to smartphones. These devices extend the capabilities of the phones by providing external sensors and interfaces for special use-cases. Even though the IoT expression is broad and not always includes a smartphone as a central component, there are still common features of IoT devices that need to be considered during software development. Hassan [10] defines IoT as:

> IoT is a world of interconnected things which are capable of sensing, actuating and communicating among themselves and with the environment (i.e., smart things or smart objects) while providing the ability to share information and act in parts autonomously to real/physical world events and by triggering processes and creating services with or without direct human intervention.

Information sharing is a key part of this definition, which suggests that an IoT device has a recipient target for their collected data. This is why IoT devices need to have a well-defined software module that manages the data sharing process between the two parties. This module can be either a separate package within a software or a standalone software development kit (SDK) for reusability.

One subset of IoT devices include smart medical devices, which consist of blood pressure monitors, thermometers, step trackers, and so on. Nowadays, most patient-oriented smart medical devices come with a smartphone application, which allows users to review their measurements over the long term.

It's less usual, however, for a specific device to allow interconnection with different applications. In 2014, both Apple and Google released a solution for sharing medical data within their mobile ecosystems; yet, these data storage options are limited.[1][2] For instance, their system currently does not store medicine related information, nor any custom raw data from a smart medical device.

If a company would like to access data directly from a device in their standalone mobile application, they require an SDK from the original manufacturer. Furthermore, with an SDK, the manufacturers can sell their devices for medical trials, where the data collection is usually managed with internal software. In general, an SDK is a valuable addition for any IoT device, as it can extend the possible use-cases of the product.

Since SDK development is a very specific and practical topic, the literature and online resources are considerably more limited compared to standard software development. The goal of this thesis is to gather some best practices for SDK development and evaluate them in an industrial context. Some of the selected techniques will be presented in the architectural design and development of an SDK, which will be created in cooperation with a business case provider company.

[1] https://www.apple.com/lae/ios/health/
[2] https://www.google.com/fit/

1.1 Background

1.1.1 Academic context

This thesis was done in the EIT Digital Academy Master School (referred to as Master School), which entailed additional constraints on the work. My selected major is Software and Service Architectures; therefore the main focus of the thesis is on the high-level design of the proposed SDK.

The Master School offers a double-degree program in collaboration with two universities. In my case, these universities were Eötvös Loránd University (ELTE) in Hungary for the first year and Aalto University in Finland for the second year. Usually, in the Master School, only the second university evaluates the thesis work, but due to the regulations of ELTE, I have to also fulfill their requirements and present my work after the submission. A software-focused thesis has practical requirements at Aalto; however, a master's thesis at ELTE is usually a theoretical scientific work. This thesis fulfills both of these requirements by providing an in-depth analysis of the SDK development practices and a well-documented implementation process.

Additionally, the Master School requires the students to collaborate with a company that provides a real-world business case as the topic of the thesis.

1.1.2 Industrial context

The business case was provided by the company Popit Oy[3] during the Autumn of 2018. Popit Oy offers a complex medical solution for medicine usage tracking. Their product portfolio includes a pill tracking hardware, two mobile applications for patients and a web service for clinical professionals.

The core component of their product is called Popit Sense, which is a smart medical device that can be attached to pill blisters to monitor pill removals automatically. It includes a wide range of sensors, which detects the movement of the pills with their patented technology. The device is presented in Figure 1.

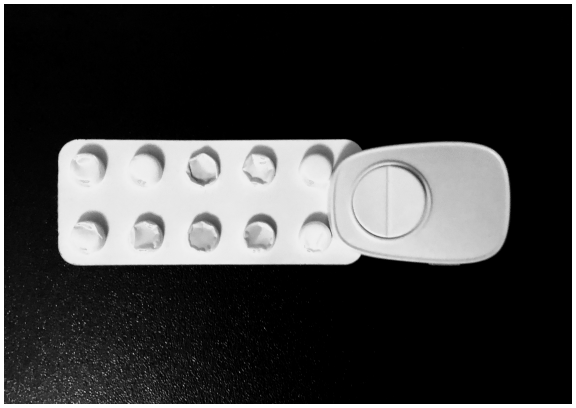

Figure 1: A Popit Sense device attached to a pill blister

[3] https://www.popit.io

1.2 Problem statement

The products of Popit are already used by patients and researchers internationally, but they are planning to integrate their solution more deeply with clinical systems. For future collaboration with pharmaceutical companies, an SDK for their device is needed, which will be the partial end-product of this thesis work. Therefore, the developed software should be of production-grade quality. The exact specification will be discussed in the next chapter, which is also summarized in the following three main non-functional requirements:

1. The SDK has to be secure.
2. Easy to integrate.
3. Future-proof.

The thesis also focuses on the general process of an SDK development, which includes the research of the recommended practices. This research will include the evaluation of a new cross-platform development technology based on the Kotlin programming language.

This new method can potentially speed up development and make the codebase easier to maintain. The thesis will evaluate these assumptions in-depth, along with the configuration and development of this new system. To measure the effectiveness of the SDK design and development process, I have defined the following three research questions:

1. **Does the architectural plan fulfil the three requirements of the SDK?** This question targets the practical solution of the SDK. The thesis will present a proof-of-concept implementation of the core features, which helps us to analyse the SDK architecture and verifies the assumptions which were made during the implementation.
2. **Is the Kotlin/Native technology stable enough for enterprise applications?** This question directly focuses on the new cross-platform development technology, which was selected for evaluation due to its promising features.
3. **What are the best state-of-the-art SDK development methods?** This question is a reflection on the whole research and development process. The expected outcome is a list of verified principles which can be used as a reference for future developers.

1.3 Structure of the Thesis

The thesis includes seven chapters. The first two covers the introduction and methodologies of the study. The third chapter presents the best practice research, which is a general overview of the subject. The fourth and fifth chapters discuss the practical solution of the business case. The gathered data from the research and implementation are evaluated in the sixth chapter, and the findings are summarized in the seventh.

2 Methodology

The three selected research questions are dependent on each other, meaning that they cannot be approached separately and the answer for one question can have implications on another question. Question 3 can be considered as the primary objective of the thesis, but it will only be answered after the selected practices have been tested with a process that first answers questions 1 and 2. Since the thesis is written in collaboration with a business case provider, the evaluation of the architectural design in questions 1 and 2 are done in a practical context. The preparation of the practical evaluation includes the analysis of the suggested best practices; therefore the hypothesis for question 3 can be made in the first half of the thesis. This hypothesis is verified with the implementation and evaluation which aims to answer the first two questions. The research process is also illustrated in Figure 2.

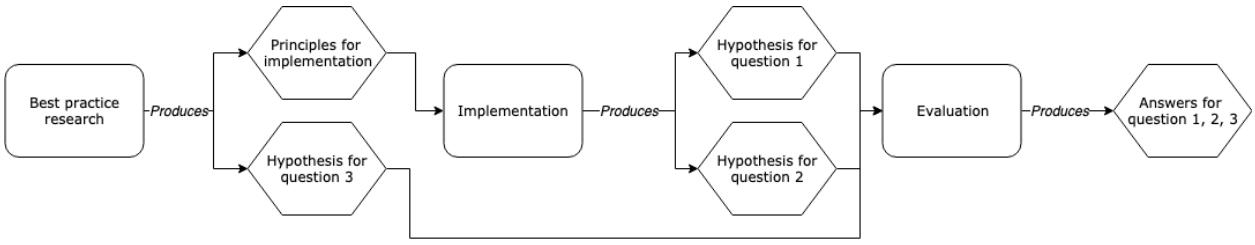

Figure 2: Research process of the thesis.

2.1 Best practice research

The research of the SDK development best practices includes a literature study, analysis of publicly available SDKs, and the cross-platform development framework selection process. This research should provide enough information about the state-of-the-art development practices to begin the practical implementation of an SDK.

Selecting materials

During the literature study, both online and offline mediums will be considered. It is expected that the number of books on the subject of SDK development is limited, as it is considered as a niche topic in software engineering. Therefore, relevant companion subjects will be included which can be utilized during an SDK development process. Some of the selected keywords and topics for searching the relevant materials: "software development kit", "software reusability", "software architectures", "software design patterns", "software development principles", "refactoring", "cross-platform development" and "mobile application development".

Determining relevance

The general approach in the study is to focus on high-level software development practices, without considering a specific programming language or platform. This makes the findings reusable for general SDK development and less dependent on modern development trends. Material that focuses too much on a specific implementation that cannot be generalized is disregarded. Some practical development

techniques that seem relevant in the development of the SDK in the given business case will also be considered.

Presenting results

The expected outcome of the research is a list of development principles. A narrower list will be created based on the common suggestions of the materials. This list should contain the relevant best practices of an SDK development process, which also serves as a hypothesis for research question 3.

Validation

Apart from the literature, the best practice research process also includes the analysis of some publicly available SDKs. The development decisions in these examples can verify the selected list of suggestions. The combination of these results also serves as the basis of the implementation for the SDK in the given business case. This implementation will be evaluated as the final step of the research. The final list of best practices will only contain the validated suggestions, which will answer research question 3.

2.2 Development

The development of the SDK for the business case provider company is the most practical part of the thesis. Detailed documentation of the process is a key objective during the collaboration with the company. The development process utilizes the list of principles collected during the best practices research. Their application is highlighted during the process, so they can be traced back during the evaluation phase to measure their effectiveness. The design and implementation of the SDK are separated in two stages, with different objectives and methodologies.

Software architecture plan

The requirement specification and software architecture definition is based on the practices of the company and the suggested framework of Rozanski and Woods [15]. The requirements will be recorded in an internal document at the company for future reference. The software architecture plan will be represented on UML diagrams and in written documentation. The company uses an agile approach to develop their software; therefore the full specification will be defined across multiple stages, and the implementation will be iterative. For easier readability, only the last version will be included in the thesis, with notes on the changes during the iterations.

Proof-of-concept implementation

For research question 2, the in-depth analysis of the Kotlin/Native technology requires the development of a working prototype. This prototype is also helpful in the verification of question 1. Due to the time limitation of the thesis work, the full plan of the SDK will likely not be implemented. For validation, a satisfactory subset of features will be selected for realization. These features will vertically go through from the highest to the lowest layer of the SDK, verifying that the technology is capable of delivering functionality on every level.

2.3 Evaluation

The development process produces two artifacts: a software architecture plan and a proof-of-concept implementation of the SDK. The functional requirements can be verified during the implementation of the plan, but the software quality assumptions need to be tested after the development. The evaluation phase combines the findings of the previous steps with additional quantitative and qualitative evaluations to answer the research questions.

Interviews

The verification of the SDK will be done with the help of additional partners. This includes developers in the business case provider company, external senior software architects, and a potential customer of the company who will utilize the SDK in the future. These parties will provide feedback on the developed software through semi-structured interviews. The questions will verify the documentation, sample codes, security, integration, programming interfaces, performance and the general experience in the utilization of the features. The results of these interviews will be used to verify the requirements and answer research question 1.

Code analysis and profiling

To answer research question 2 the implemented SDK requires additional testing. The source code will be investigated with a static code analysis tool to find potential issues in the implementation. A profiler will also be used to measure the execution of the software and quantify the performance metrics. The stability of the SDK will be measured through manual testing of the sample applications, as they require a hardware component to access all functionalities. If the result can meet the requirements and the source code is easy to maintain, then Kotlin/Native can be considered as a viable option for similar use-cases.

Comparison of best practices and development

Finally, the development process will be evaluated as a whole. The hypothetical list of development principles will be checked through the practical decisions of the process. The structure of the implementation needs to be mapped to the architectural design plan to verify the design decisions. A selected principle will be verified if it was successfully used during the development. If it was used without success, the principle is rejected. If it was not used, that principle could not be verified in the given business case. The verified list of principles will serve as an answer for research question 3.

3 Best practices research

3.1 Literature study

The literature study contains both offline and online materials, including books, thesis works, conference presentations, blog and forum posts. The number of books about SDK development is relatively small since it is a niche topic compared to full-featured software development. Some books about software architectures or mobile application development cover this topic as a subsection. In those cases, the authors are not going into details about the subject. However, many mainstream subjects can be associated with the parts of the development process of an SDK. For example, cross-platform development and refactoring are both essential topics in our business case; therefore the literature study included additional books about these subjects.

Compared to books and academic literature, there are many more online articles about SDK development practices. This is likely because the development process is mainly discussed in a practical context in a specific application. These articles are usually focusing on one platform and one programming language, with implementation examples. This literature study tries to select the platform independent and language agnostic best-practices of an SDK development, which can be extracted from the practical tips that are presented by these articles.

3.1.1 Key concepts

API and SDK differences. Hardware devices that share data with mobile applications offer their integration with an application programming interface (API) or a software development kit (SDK). In most cases if a hardware manufacturer only supplies an API for their device, they refer to a web API, meaning that the hardware data indirectly comes through the internet. One practical example is the API of the Oura[4] smart ring device. This might seems counter-intuitive; however, it is a reasonable approach for a company which needs to ensure that they will gather every available data that is coming from their product, even if it is integrated into a third-party solution.

Using a web API means that the application cannot access the hardware data without internet, which is not a suitable solution for Popit, nor their partner companies. Sandoval [16] defines an SDK as: "An SDK is a full-fledged workshop, facilitating creation far outside the scopes of what an API would allow." By building an SDK, we can supply every necessary software component to directly collect the data from the device.

However, this difference does not mean that an API will not be created. Some of the public SDK documentations purposefully avoid using the word API to describe the interface of the SDK to prevent confusion with web APIs. Still, every SDK that exposes public functions to a host application does it through a local API. The

[4] https://cloud.ouraring.com/docs/

implementation chapter of this thesis will present the design and development of an API for the SDK.

Software framework and library differences. Framework and library are both common words to describe a set of resources for software development, but the difference between the concepts is not always clear, and it can depend on the context. Commonly online articles state that the difference comes from the inversion of control (IoC). Wozniewicz [23] explains it in the following way:

> The technical difference between a framework and library lies in a term called inversion of control. When you use a library, you are in charge of the flow of the application. You are choosing when and where to call the library. When you use a framework, the framework is in charge of the flow.

However, this is not always true in every implementation. Apple Inc. [2] simply defines frameworks as a "hierarchical directory" and states that "frameworks serve the same purpose as static and dynamic shared libraries". In their development toolset, a framework can be selected as a target output for iOS or macOS, which does not enforce IoC, but can act as a bundle for more resources than a library.

The definition of an Android library in the documentation by Google LLC [8] leads to further confusion. They state that an Android library can act as a bundle with additional resources next to the source code. This implies that their definition of a library is very similar to the definition of a framework on iOS platform. Additionally, they do not specify frameworks in their development documentation and their development toolset there is no different targets for frameworks or libraries.

In practice, IoC can be implemented in library outputs both on iOS and Android. If we accept the commonly used definition, we can say that even if the output is labeled as a library, it can be considered as a framework if IoC is included in the implementation.

Alternatively, some authors[19] only consider the scope of the implementation as a difference. From this perspective, a library is a set of functions that can be used in the source code without any additional setup. An example could be a mathematical library that provides calculation methods. A framework, on the other hand, needs configuration before use, which is a typical case for SDKs.

Regardless of the definition, it is clear that a framework is as a superset of a library. We can consider an SDK as a framework if either the IoC implemented or there is additional configuration required for active usage in the host software.

Static and dynamic library differences. The delivery process of a mobile SDK requires the discussion of the difference between a static and a dynamic library.

The source code of a static library is entirely loaded into the memory during runtime based on the development documentation by Apple Inc. [1]. On the other hand, dynamic libraries compiled separately and only loaded in the memory when they are referenced. In this documentation, static libraries are mentioned as an

inferior alternative. Since 2014 Apple introduced dynamic library support on iOS, and they highly encourage its usage for performance improvements.

The difference between static and dynamic libraries are defined the same way in C/C++ sources, which can also be used in Android. However, most of the Android applications are using a JVM based language (generally Java or Kotlin) which dynamically loads the classes from the libraries by default.[12]

Design patterns: Observer, Facade, Singleton. There are multiple different software design patterns used during the specification of the SDK. Most of them are well-known, but their exact definitions need to be introduced before the discussion of the architectural decision. The most important patterns in this thesis work are:

- **Facade:** A Facade is a wrapper around multiple features in a software package, usually in the form of a static class. This pattern is frequently used in the implementation of an API. In many cases, this is the only interaction point with an SDK, as it helps to hide the underlying mechanism of the different modules.

  The definition by Vlissides et al. [22]: "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."

- **Observer:** A commonly used design pattern in the data management of an SDK. An Observer is a specialized interface, which can be used to broadcast information from the SDK to the host application. It is especially useful in cases where the data processing is done asynchronously.

  The definition by Vlissides et al. [22]: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

- **Singleton:** Perhaps the most widely used pattern in software development. A Singleton is a constrained class that only allows the initialization of one instance, which can be accessed through a static variable. It is not explicitly used in SDK development, but some part of the implementation depends on it.

  Kotlin, the selected programming language of the development uses companion objects instead of static classes, which are essentially functioning as Singletons.[5]

  The definition by Vlissides et al. [22]: "Ensure a class only has one instance, and provide a global point of access to it."

3.1.2 SDK development principles

Business logic separation. Some smart devices begin their development cycle without an SDK in mind, and their features can be only accessed through a full-featured application. These applications usually include every resource that could

[5] https://kotlinlang.org/docs/reference/object-declarations.html

potentially be transformed into a standalone SDK, but based on their design decisions, it might be hard or impossible to separate them from the host software.

If a mobile software architecture is well-designed, we can prepare the development of an SDK even if we only develop a standalone application first. Eidhof et al. [6] presents several good examples of modern mobile architecture standards. Not all of these are usable for an SDK development, as they are focusing on the whole application logic, including the graphical user interface (GUI) and user interaction handling. However, the chapter about networking discusses an example of designing a component without a GUI. They introduce the "controller-owned" and "model-owned" approaches, which are both popular solutions in application development. Structuring an application logic in "model-owned" fashion can be a significant first step in building an SDK, as it makes it easier the separate the business logic from the higher level components.

Best practices. Apart from traditional literature, the research included exploration for best practices on the internet, which are shared by experienced developers. This resulted in the selection of four relevant articles, which include a list of suggestions for effective SDK development. Srivastava [20] presents 19, Smales [18] 36, Levinsky [11] 9, and Robosoft Technologies [14] 14 principles. From these articles, I have selected the common guidelines, and with the combination of the results from the whole literature research I have narrowed the findings to 8 main principles, which can serve as a guideline for an SDK development:

1. **Aim for simplicity.** The first and most important principle, which is supported by all of the collected literature. Since an SDK is an internal component of a larger software, keeping it simple is beneficial for both the integration and maintenance. Usually, an SDK has one specific task, and it should not include any additional features that are not used by the host application. Developers commonly tend to choose a different alternative if they feel that the API is difficult to use or it has too much overhead for their use-case.

2. **Use minimal external dependencies.** A special note on simplicity refers to the usage of dependencies. A general suggestion for software developers is never to reinvent the wheel and use existing modules to solve common problems. This suggestion is not as straightforward for SDK development, since using external dependencies can significantly increase the footprint of the software, and could lead to compatibility issues with the host application. Therefore, this point suggests that only use dependencies if they are a critical component in the SDK.

3. **Minimize resource usage.** An SDK can affect the CPU, memory, and network usage in the host application. This should be kept in mind during the implementation when allocating new resources in the source code. It is recommended, to use a profiler for measuring the footprint of the SDK after every new release.

4. **Minimize permissions.** On mobile platforms requesting permission for a system level feature is usually an opt-in process with the interaction of the end-user. Therefore, the fewer features required for an SDK, the easier it is to integrate into an existing application. Unexplained permission requests can also lower the trust in the customers of the software.

5. **Keep security in mind.** Security risk in an SDK can act as a loophole in a host application for adversaries. Therefore, the standalone security of an SDK is as vital as the security of the whole software. Sensitive information cannot be included in the source code, which might be accessed by the customer.

6. **Create sample codes and documentation.** Providing sample code snippets for the integration and main features of the SDK is the easiest way for developers to learn their usage. Every public classes and function need to be documented. A change log should be included to present the differences between the versions.

7. **Design for the long-term.** From the point when an SDK is integrated into a system, the development life cycle has to consider the host application too. Developers need to think ahead in the design of the API and prepare the codebase for long-term maintenance. It is better to add extra features than removing them, but it a feature needs to be deleted, the developers should always note the deprecation first and roll out the destructive changes in the next version.

8. **Get feedback from developers.** Since software developers utilize an SDK, the audience is more than capable of providing useful feedback on the product. It is essential to reach out and interview developers about the documentation, integration, and overall functionality.

3.2 Analysing public SDKs

Software development kits are usually built exclusively for business customers, and they are only shared with their target groups after they made a non-disclosure agreement with the SDK supplier company. Fortunately, there are still several public SDKs available for consumer applications.

The research included the analysis of several hardware and software SDKs, which were used as examples to design the API, documentation, and overall architecture. The source code of these SDKs can be only accessed in a few cases; therefore the analysis cannot cover the internal structure of the selected samples.

Most of the selected SDKs are only providing a minimum feature set for the end-users. They are hiding the low-level mechanism, and their API presents a high-level abstraction. The SDKs which includes some background data management functionality require the host application to initialize a global instance of their software in the root of the project. (This initialization is usually done in the MainApplication class on Android and AppDelegate on iOS.) Some APIs which has some asynchronous operations are using shared interfaces to return results. For example, if a device

connects to the phone, the SDK calls one of their implemented interfaces to notify the host application about the event, with some additional supplied data about the state of the device. The following list briefly introduces the selected SDKs and highlights some of their unique solutions:

1. **Estimote**[6]
   Estimote makes iBeacon devices, which can be used for indoor location tracking. Their SDK is an excellent example of clear and detailed documentation. Even though Bluetooth communication requires a complex low-level configuration, their API hides the internal mechanism and presents an easy to use abstraction.

2. **DJI**[7]
   DJI is well-known for drone and camera products. Their public SDK enables the development of custom applications for their product portfolio. Additionally to the documentation and sample codes, their website also presents an architectural overview of their solution. This useful example shows how they structured their underlying components and how they grouped the features in abstract containers.

3. **Sphero**[8]
   Sphero produces small robots, which can be controlled wirelessly with a mobile application. Since they offer a wide range of complex features, their documentation is split into a basic and advanced part for different target audiences. Unlike the other samples, Sphero also provides documentation for their low-level data packets in the communication protocol. This information allows developers to utilize the full potential of the hardware.

4. **Philips Hue**[9]
   Philips Hue is a smart LED light system with color-changing features. Their developer website presents a long list of supported programming languages, thanks to their flexible web API and the support of the open-source community. Apart from the API documentation, Philips also presents design considerations for building an application with their devices, which includes tips on the user experience and hardware-specific development caveats.

5. **Square**[10]
   Square makes a mobile bank card reader, which physically connects to a smartphone. The integration of their framework is relatively more complicated compared to the other examples, as the full payment management flow needs to be set to make use of the hardware. Fortunately, their documentation is straightforward, presented as a step-by-step guide for the different functionalities of the device. Even though they do not disclose information about the security

[6] https://www.estimote.com
[7] http://www.dji.com
[8] https://www.sphero.com
[9] http://www.meethue.com
[10] https://www.squareup.com

considerations of the SDK, some features are purposefully limited to prevent misuse. These limitations are highlighted in their documentation with an explanation.

6. **Notch**[11]
   Notch is a body movement tracking hardware that directly sends the collected data to a mobile application. The software can connect to multiple Notch devices at the same time, which are placed on different parts of the body. Based on this data the software can represent a three dimensional model of the movement. Therefore, the SDK offers a high-level object-oriented representation of the collected data, through the Bone, Skeleton, Workout, and Measurement classes. Their documentation not only contains sample codes, but also a template application which includes all the necessary configuration for a quick start with the device.

Comparison of the SDKs

The features of the analysed SDKs are summarized in Table 1. This comparison does not hold any negative implication about the different development styles, but it can be used to find commonly used practices. For example, the inclusion of hardware documentation is a valuable feature that was not suggested by the analysed literature. Using an observer pattern is a common solution for handling asynchronous behaviors, which also suggests an idea for an API implementation. The presence of a low-level API and the separation of the core features can be considered as an edge case; therefore they are not included in all examples. On the other hand, the user experience guideline would be a useful addition for all commercial SDKs to ensure that they are used in the right context.

[11] https://wearnotch.com

	Estimote	DJI	Sphero	Philips Hue	Square	Notch
Requires API key	-	✓	-	-	✓	✓
Low-level API	-	✓	✓	-	-	-
Simple and advanced integration	✓	-	✓	✓	-	-
Uses observer pattern	✓	✓	✓	✓	-	-
Sample application	✓	✓	✓	✓	✓	✓
Hardware description	✓	✓	-	✓	-	✓
Software architecture specification	-	✓	-	-	✓	-
Core features in multiple packages	✓	-	✓	-	-	-
Object-oriented API	✓	-	-	✓	-	✓
User experience guidelines	-	✓	-	✓	-	-
Requires extra permissions	✓	✓	✓	-	✓	✓
Troubleshooting	✓	✓	-	-	✓	✓
Changelog	✓	-	-	-	✓	✓

Table 1: Comparison of the distinctive features of the selected public SDKs.

3.3 Development framework selection

The product development of the Popit Sense device is still an ongoing process; therefore the set of functionalities is continuously expanding. Updating the implementation to support the new features is always a difficult task, especially if this includes low-level data processing. Managing these changes on multiple platforms is expensive and could lead to fragmentation in software quality. Fortunately, there are many different tools available to develop cross-platform mobile applications, and some of them can reproduce the functionality of the existing applications. Before we selected a specific framework, we had to analyse the available options.

We started this analysis by categorizing the mobile cross-platform development tools based on their target software layers. In software development communities these tools are simply referenced as cross-platform development frameworks, even though their feature sets are significantly varying. This categorization is helpful to select the appropriate tool for specific use-cases:

High-level focused:

Arguably the most popular category of cross-platform development tools, which includes React Native[12], Cordova[13], and many other frameworks. The purpose of these frameworks is to speed up the development of GUI and data manipulation

[12] https://facebook.github.io/react-native/

[13] https://cordova.apache.org/

focused applications. Even though some of these frameworks include a wide-range of operating system (OS) level APIs, they do not support direct interoperability with existing native source code. Additionally, in case of React Native, the official supported programming language is JavaScript, which is not suitable for low-level programming either on iOS or Android.

Low-level focused:
Traditionally, low-level libraries are developed in C/C++ as it is supported on most operating systems. It is the best option for high-performance software since the language can utilize the OS level APIs without any overhead. The downside, however, is the hard maintainability and difficult memory management. The lack of GUI support in our case is not a disadvantage. An alternative to C/C++ is the GoMobile framework which allows cross-platform mobile development in Go language. Even though it is a lot easier to maintain a Go-based project, the surrounding developer community is a lot smaller around their mobile development solution compared to C/C++. This implies that there are a lot less third-party libraries which could support a development process, which makes it a worse alternative than C/C++ for larger scale projects.

Full-featured frameworks:
The smallest category from the three, which combines both high and low-level software development of cross-platform applications. One popular example is Xamarin[14], which includes the Android and iOS OS level APIs with a C# bridge. This interface allows developers to write fully cross-platform applications in one language and with a unified GUI. Even though native libraries can be included in a Xamarin project by their bridging mechanism, this process is only one-way, meaning that there is no easy way to produce native libraries with Xamarin.

From the first glance, we can see that the SDK should be developed with a low-level focused cross-platform framework, as the other options are not providing enough features to meet the specification. Apart from C/C++, recently a new technique has surfaced in the form of Kotlin/Native, which provides the same low-level feature set, but with automated memory management and in an easily maintainable language. Even though this technique is very new and might not be entirely production ready, the business case provider company took the risk to improve the long-term development efforts potentially. A brief risk assessment can further verify the choice: since Android fully supports Kotlin, even if the Kotlin/Native project fails to deliver the full cross-platform implementation, the SDK will still be usable on one platform.

For this reason, we have selected Kotlin/Native for further evaluation with a proof-of-concept implementation.

[14] https://visualstudio.microsoft.com/xamarin/

4 Software architecture design

The design process of the software architecture is the first collaboration point with the business provider company. It is separated into three stages:

1. **Requirements specification:** The functional and non-functional requirements of the SDK are discussed through a series of meetings with the stakeholders of the company. These requirements are used in the design of the architectural plan as constraints and feature definitions.

2. **Codebase analysis:** Part of the SDK is built on top of an existing codebase, which was originally made for Android and iOS systems. This code is analysed in-depth, which also serves as a functional requirement for the features of the SDK.

3. **Architectural plan creation:** Finally, the plan of the software architecture is created from the defined requirements. The plan includes different perspectives and abstraction levels, which describe both the external and internal structure of the developed system.

4.1 Requirements specification

A software requirements specification process is always a challenging task. A false assumption about a planned feature can affect the whole development life-cycle, and the end result as well. This not only includes functional errors in the system but also an improper execution of a business feature, which can reduce the value of the product. The main challenge in the specification process is the communication between the stakeholders, who are a mixed group of people with different backgrounds and technological skills.

Existing product. Since the SDK is based on the functionality of two existing software, the first step in the requirements specification was the analysis of their features. The two software provide the same functionality on iOS and Android mobile operating systems. The actual implementation of their codebase will be discussed later in this chapter. From a high-level perspective, the main features of the application can be summarized with the following set:

- The application allows users to track the usage up to three medicines, with or without an associated Popit Sense device.

- The medicines can be entered in the software along with their reminder times. These reminders can be set to be repeated on every day or specific days of the week.

- The application provides a visual guide for pairing the Popit Sense device with the software. The user first needs to select a medicine, select the pairing menu, press a button on the Popit Sense device, and if the process is successful, the device will be associated with the selected medicine.

- The application displays information about the Popit Sense device, including their serial number and battery level. This data is updated whenever the device synchronizes.
- The Popit Sense device recognizes if the user took a pill from a blister. This pill taking event is automatically synchronized with the application in the background.
- The application only presents the reminder for the users if the Popit Sense device was not reporting a pill taking event at the time of the reminder. This reduces the unnecessary amounts of reminders if the pill was taken correctly.
- If the users agreed to share their data for analytical purposes, the application sends the stored pill taking events to a web service.
- The application also able to receive data from a web service, which can contain remote configurations for local features.
- The software automatically updates the firmware of the paired Popit Sense device in the background.

It is essential to keep in mind, that the new system is a standalone product, even if it is mostly based on the existing features. The original applications were designed with a graphical user interface, which is interconnected with the hardware functionalities. The SDK will only contain a programming interface; therefore the representation of the features might use a different abstraction in the new system.

Use-case of the SDK. Since third-party companies will utilize the SDK, it is vital to take their requirements in consideration for the software design. The target audience is business customers, who will utilize the Popit Sense device in their clinical trials. In the future, the customers might also include different health-management applications, who are providing business-to-consumer style solutions. Security, long-term support, and easy integration are significant aspects for both targets.

A clinical trial in some case can last for years, where the software specification cannot be changed to preserve the same settings through the research. If the SDK stops functioning during this period, the company either has to pay a fine for publishing the modifications or the device will be excluded from the trial. This constraint makes future-proofing a critical part of the SDK.

During my collaboration with the business case provider company, I have taken part in a meeting with a potential customer. This discussion demonstrated that the feature set of the SDK is a valuable asset that can be extended to support specific use-cases. The customer has a production-ready application with an internal medical data management implementation. The case provider has offered a middleware, which can be part of the SDK bundle, that transforms the collected pill events to a format of the host application for easier data processing. A long-term plan for the SDK is the development of optional features for companies, which can be included based on the build configuration.

Functional requirements. The functional requirements are partially based on a subset of the previously mentioned features of the existing applications. This set is extended with additional functionality related to the use-case of the SDK. The following set presents the specified functional features:

- **Device pairing** The first interaction point in the mobile application with a Popit Sense device is the pairing process. This operation is originally managed through a GUI; which needs to be transformed into a programming interface in the SDK. Intuitively, we expected that the underlying code of the graphical interface could be reused as an implementation for the SDK. Later we will see, that in practice this is harder to achieve because of the abstraction of the existing solution.

- **Medicine input** The pairing process should always require the input of a medicine name. Pairing the hardware with the application does not require a medicine name on the source code level, but this is vital information for future user analytics.

- **Device data synchronization** The main purpose of the SDK is to read all data from the paired devices and present it to the host application in a processed format. The SDK should provide public high-level classes of the processed device information and pill taking events. These should be automatically read whenever the device sends the data, even if the software is working on the main thread or the application is in the background.

- **Device management operations** The SDK should provide public methods to manage the paired devices. These methods should include the unpairing, sensitivity level changing, and utility information reading of a device.

- **SDK status reporting** The operation status of the SDK should always be reported to the host application. The exact list of status codes and exception types will be discussed in the implementation chapter.

- **Firmware update** The SDK should update the firmware of the paired devices automatically, without any user interaction.

- **Data journaling** The SDK should store all collected data from the devices. However, it should not allow the host application to access the persistent storage of the pill usage data. This forces the host applications to build their own abstraction around the collected pill usage events, which makes the system easier to maintain.

- **Web service communication** The SDK should be able to send the collected data for a web service if the analytical data collection is allowed by the host application. This process should be automated and executed in the background, with a limited amount of retries, if the data transfer was not successful.

Non-functional requirements. The discussion of the functional requirements of the SDK was straightforward, as they are based on existing implementations. The non-functional requirements, on the other hand, requires more in-depth analysis, since these are standalone features of the SDK. Through a series of meetings with the business provider company, we defined the three main quality properties of the SDK:

- Security: Popit Sense is a classified medical device; therefore security is a critical aspect of the development. The SDK should not allow access to the underlying implementation for the developers of the customer company. It should not store any sensitive information in the source code, such as API keys or passwords. If possible, the reverse engineering of the SDK should be prevented or hardened. A customer-specific key for the SDK should also be considered to avoid the unauthorized usage of the software outside of the contracts.

  The vulnerability of the system is analysed and tested against the OWASP Mobile Application Security Checklist.[15] The additional security considerations will be evaluated through interviews with senior developers.

- Easy integration: The system will be used by third-party companies, who need to be able to include the SDK in their existing software without any difficulties. The API, documentation, and sample codes of the SDK will be based on the public examples of well-made SDKs. The unexpected difficulties in the integration process will be eliminated iteratively through meetings with the first customer.

- Future-proofness: The long-term plans of the SDK suggests that the business case provider company will use it internally to add new features to their mobile applications. Because of this, the future development of the SDK should be easier than their current single-platform applications. The future mobile OS versions should also be supported, although it might not be possible to guarantee at this point.

  The implementation process in this thesis will only cover the primary data management features; therefore some parts of the functional requirements will be implemented by other developers. For this reason, the codebase should be easy to maintain, regardless of the complex cross-platform configuration.

Summary. To analyse the collected requirements, we have applied the viewpoints concept from Rozanski and Woods [15]. The book introduces viewpoints as a reference for the quality requirements of an architecture, from the different structural aspect of the system. We have applied the viewpoint catalog of the book in our case:

1. **Functional:** The feature set of the SDK should match the existing mobile applications features, which is presented through a high-level API for easy

[15] https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide

integration. The exact specification of the functions will be built into the architectural plan.

2. **Information:** The SDK should store all incoming data from the hardware and forward them to a cloud database. This database should be stored securely, and the API should not allow direct access to the raw data for the host application.

3. **Concurrency:** The SDK should function in parallel with the host application's features. It should continue monitoring for devices until the application is terminated.

4. **Development:** The SDK should be developed as a cross-platform project in one version control repository. Every source code that is not platform-specific should be written in a shared module to prevent duplication.

5. **Deployment:** The output of the release build should be a self-contained framework file, which can be integrated easily in the host applications. This should be shared in a separate version control repository, which should also contain the documentation of the API.

6. **Operational:** The operational specifications will be described with the software architecture model, along with the list of the supplied features.

4.2 Codebase analysis

The analysis of the mobile applications of Popit is a crucial part of the software architecture design process. Part of the new implementation will directly reuse the lines of the existing codebase or builds upon them with slight modifications.

I started my work at the company several weeks before the SDK development started. This allowed me to get familiar with the codebase of the iOS application and the different operations of the Popit Sense device. In theory, the Android application should have provided the same feature set with the same software architecture, yet in practice, the implementation had many differences. Since the Android version was created later, it had a slightly cleaner implementation. Additionally, since 2017 Google officially added IDE level support for Kotlin for Android development[16], which also included a Java to Kotlin code converter tool. These reasons make the Android application a reasonable basis for the software design of the SDK.

The codebase of the existing software analysed from the highest to the lowest levels. This results in an overview of the existing architecture, and an overall impression of the implementation decisions. The analysis only focuses on the Popit Sense device related functions in the applications, as the other features are not included in the SDK.

[16] https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/

4.2.1 High-level overview

Abstraction layers. The implementation of features in the application can be separated into different abstraction layers, which are described in the following list:

1. **GUI:** The user interface layer of the features, which is omitted from the SDK implementation. However, the classes of the GUI includes methods for invoking the device management processes, and some of these needs to be bundled in the SDK.

   The Android implementation utilizes the platform-specific Fragment classes to display a set of user interface elements on a page. For example, the DeviceFragment class is responsible for the device pairing process and utility information display. The initialization of the pairing process is linked to a button, which forwards the request for the underlying business logic.

2. **Device management:** In the Android implementation, the device data management related source code are included in the same package. The use-case of the classes as follows:

   - **DeviceManager:** As the name suggests, the DeviceManager class is the main component of the device management features. It is defined as a singleton, and upon initiation, it also starts the low-level Bluetooth communication procedures. The manager listens to the events of these procedures. These events are always specific for a paired device. Based on their results, the manager initiates a DeviceInfoSyncer class with the identifier of the device. This object will handle further data synchronization procedures.
   - **DeviceInfoSyncer:** The DeviceInfoSyncer schedules the information synchronization between the Popit Sense device and the mobile application. The flow of the operations are based on the list of messages that are coming from the MessageManager class, and the DeviceOperation handles their execution. Both of these classes are based on the device identifier that is associated with the DeviceInfoSyncer instance. Meaning, that the synchronization process is always managed separately for every paired device.
   - **DeviceOperation:** This class is middleware between the low-level Bluetooth communication modules and the synchronization process. It contains operations to read-write data, and to download or upload files to the devices.
   - **MessageManager:** The MessageManager is a middleware between the persistence layer and the synchronization process. The application uses this class to insert different operations into the database, which then read in a first in, first out manner. The class also contains some predefined operations that can be added to the queue, including serial number, firmware version, and battery level reading commands.

- **DeviceConstants:** A helper class which includes the necessary constants for the communication process. In the low-level implementation, the operations are defined by integers, which are mapped to their meaningful counterparts in this class.

3. **Persistence:** The collected device data are persisted in the local storage within the mobile application. The persistence layer is built on top of an Android and iOS specific database management module. The device data is mixed with the different application data in the storage, which makes it harder to separate from the current implementation. The SDK needs to replicate the device information and pill event storing mechanism of the current application, in a shared component between the two platforms if possible. The data classes these elements are used as the basis in the new implementation.

4. **Low-level communication:** The lowest level of the implementation is a standalone module under the name of BluetoothStack. This component was developed in parallel with the development of the hardware. It includes every necessary feature to handle the communication between the application and a Popit Sense device. Since this module heavily relies on platform specific libraries on iOS and Android, it cannot be refactored to be a shared component in the SDK. Instead, the whole native module will be reused as a subcomponent in the bundle, which ensures that the original and verified implementation is used in the platforms. As the hardware provides the same feature set in both cases, the input and the output of the BluetoothStack can be bridged in the shared module of the SDK.

Code visualization. The analysis of the SDK was done both manually and with the help of code visualizer tool called Sketch It![17]. The visualizer tool was only used in the Android application since the package structure was mostly the same in both platforms. The output image (Figure 3) contained every class from the Android application. It is difficult to see the class-level relationships on the image, but the connections between the packages can be traced back. The tool also generates a text-based chart definition, which can be used to check the relationships one-by-one. The diagram did not provide clear information about the structure, but it helps to verify which classes were using the hardware data management classes.

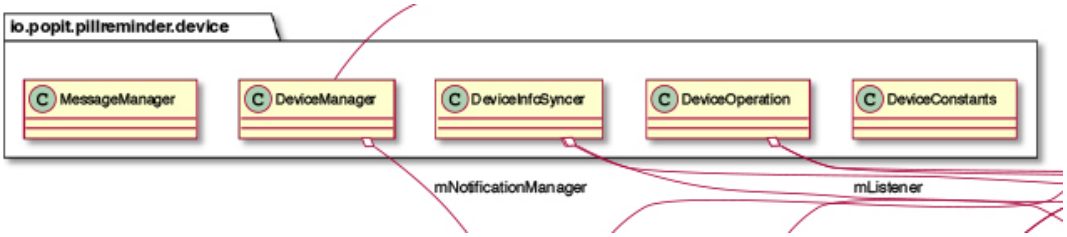

Figure 3: Excerpt from the visualized relationships of the classes.

[17] https://plugins.jetbrains.com/plugin/10387-sketch-it-

4.2.2 Source code overview

The in-depth analysis of the source code is done in parallel with the implementation of the SDK. This may sound counter-intuitive, but it is a conscious decision to improve the quality of the new shared classes. Without too much influence from the original implementation, the SDK functions as a clean slate and allows us to create a better abstraction. Parts of the implementation is exact reuse of the original source code if a significantly better solution is not available. It is important to note that the source is not guaranteed to be error free. Some hidden issues might get included in the new implementation, even with the careful remodeling of the system.

Reused classes. The main focus in the analysis of the classes is the portability of the source code. Even though Android Studio allows the direct conversion of Java to Kotlin, not all language features are available in the shared codebase for both Android and iOS. The exact limitations of the shared component will be discussed in the implementation chapter. Since the SDK development can be considered as a refactoring process, the code quality of the existing implementation needs to be verified. The related anti-patterns[3], code smells, and refactoring principles[7] are noted in parentheses next to the highlighted issue. The general notes about the portability and code quality from the perspective of the reused classes:

- **DeviceManager:** The Android implementation of the class utilizes an Observer pattern[22] to advertise the status of the connected devices for the rest of the application. Part of the BluetoothStack is initialized in this class, which puts too much responsibility in one file. (Feature Envy[7]) On the other hand, many device management features are missing from the original class, including the queries and deletion of paired devices. These features are directly accessed through the persistence layer, which is a wrong abstraction, as it makes the GUI classes too deeply connected with the lower layers.

  Refactoring notes: In the new implementation all the device management features should be included in the DeviceManager class, which helps the reusability of the frequent device management operations. (Replace Temp with Query[7]) The life cycle of the BluetoothStack should be managed in a standalone object, and the DeviceManager should only listen to the relevant events to the paired devices. (Extract Class[7])

- **DeviceInfoSyncer:** The DeviceInfoSyncer class provides only one public function to start the data synchronization for a specific device and an interface to advertise the progress. Since the class schedules the full communication process, it includes many helper functions to process the incoming and outgoing data. This makes the original class long and hard to read. (Large Class code smell[7]) On top of that, the scheduling function is executing many different asynchronous functions, which waits for their execution through callbacks from their interfaces. This means that all of these functions are nested in each other, which makes any change in the process a complicated task. (Stovepipe System[3])

Refactoring notes: Separate the data processing helper functions into a new class. (Extract Class[7]) Recreate the operation scheduling with sequential process manager to eliminate the nested functions and make the future modifications easier.

- **MessageManager:** The implementation of the class is based on a simple queue model, which is directly stores the data in the persistence layer. The name of the class is slightly misleading, since the management is related to all device communication operations, which are not only messages.

  Refactoring notes: The new implementation will be called as OperationManager, and the underlying data class for the scheduled operations will be named as Operation. (Rename Method[7])

- **Data classes:** Most of the device data classes will be reused without major modifications. The storage of the device utility information and the pill taking events are separated into two different classes. The DeviceInfo class includes mostly static values, including the serial number, firmware version, and battery levels. The pill taking events originally stored in a high-level object called PillData, which can be linked with the associated medicine. In the SDK implementation, this class is renamed to DeviceEvent, as the incoming pill taking events can also contain additional data apart from the pill usage. The original class was used to display the pill usage on the user interface; therefore additional processing was applied on the collected data. The SDK should only forward the raw events to the host application, which makes the DeviceEvent a simplified version of the PillData class.

Platform differences. The original approach for the mobile application development required the implementation of the two codebases separately, meaning that every modification should be applied twice. This can lead to unexpected differences in the two platform, even with a very careful specification. Additionally, the low-level differences between the two operating systems require an entirely different solution for a specific problem in the two cases.

These differences cannot be avoided even with the shared codebase, but they should be noted for reference in the SDK development. The general approach for the SDK development is to create every class in the shared module if possible, and bridge the platform-specific solutions with the same interface.

The most notable difference between the Android and iOS application is the contrast in the code quality. Since the Android version was built based on the design of the preexisting iOS application, the implementation is significantly more mature and easier to maintain. One practical example is the management of the device related constants, which are clearly separated in a helper class on Android but inlined in the operations on iOS. (Magic Number[7])

Functionally, the two software is very close to each other. All device management feature can be reproduced on both platforms. One important difference in the pairing process is an additional permission requirement on the Android side. After Android

version 6.0 the permission for accessing location data is an opt-in process; therefore it has to be requested from the user.[18] On Android, the coarse location permission is required to scan for Bluetooth devices in a local area. This is not needed on iOS, as they provide a separate API for retrieving devices.

The difference in the persistence layer and the data classes are also noticeable. The naming convention of the variables and property names are very loose, and in some cases, the same property is described with an ambiguous name in the two implementations. The properties of the data classes will be merged in the common code of the SDK, and the new field names will be based on the version which describes the functionality better.

4.3 Architectural plan

The requirements combined with an overview of the existing codebase provided sufficient information to define the high-level plan of the SDK. These plans not only served as the guidelines of the development, but also were used to verify the specified requirements. As a recommended software development principle by Davis [5] (Principle 48), the requirements are made with multiple viewpoints, and they are represented in separate diagrams. In our case, we utilized a component diagram to visualize the largest logical blocks and an activity diagram to present the features of the device management.

[18] https://developer.android.com/training/permissions/requesting

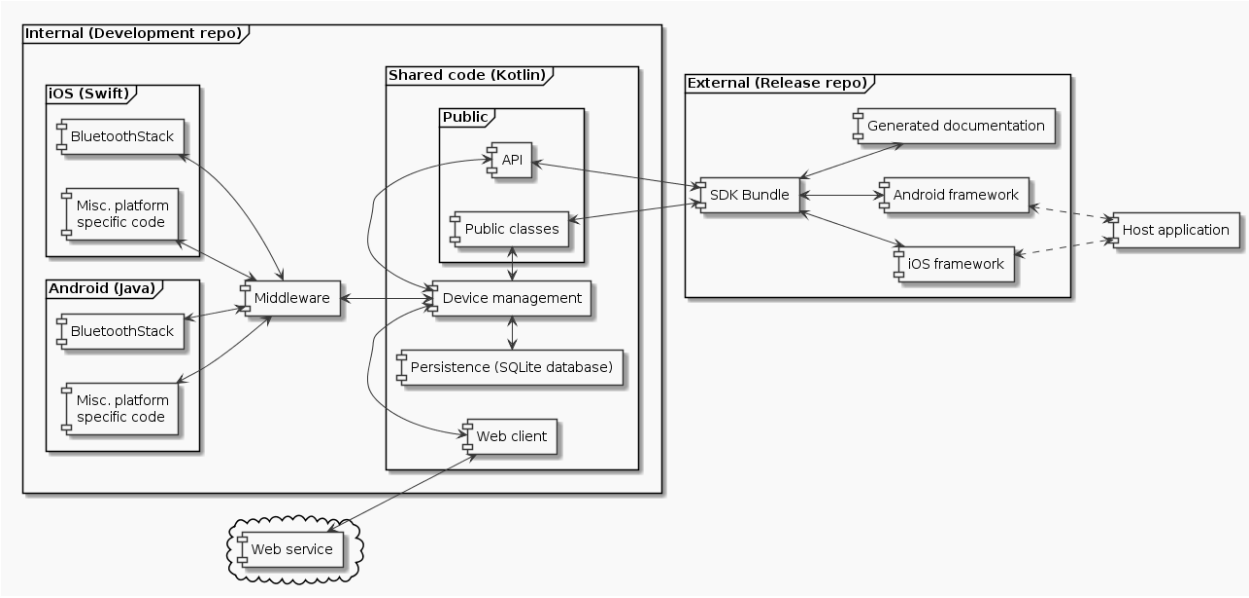

Figure 4: Component diagram of the SDK

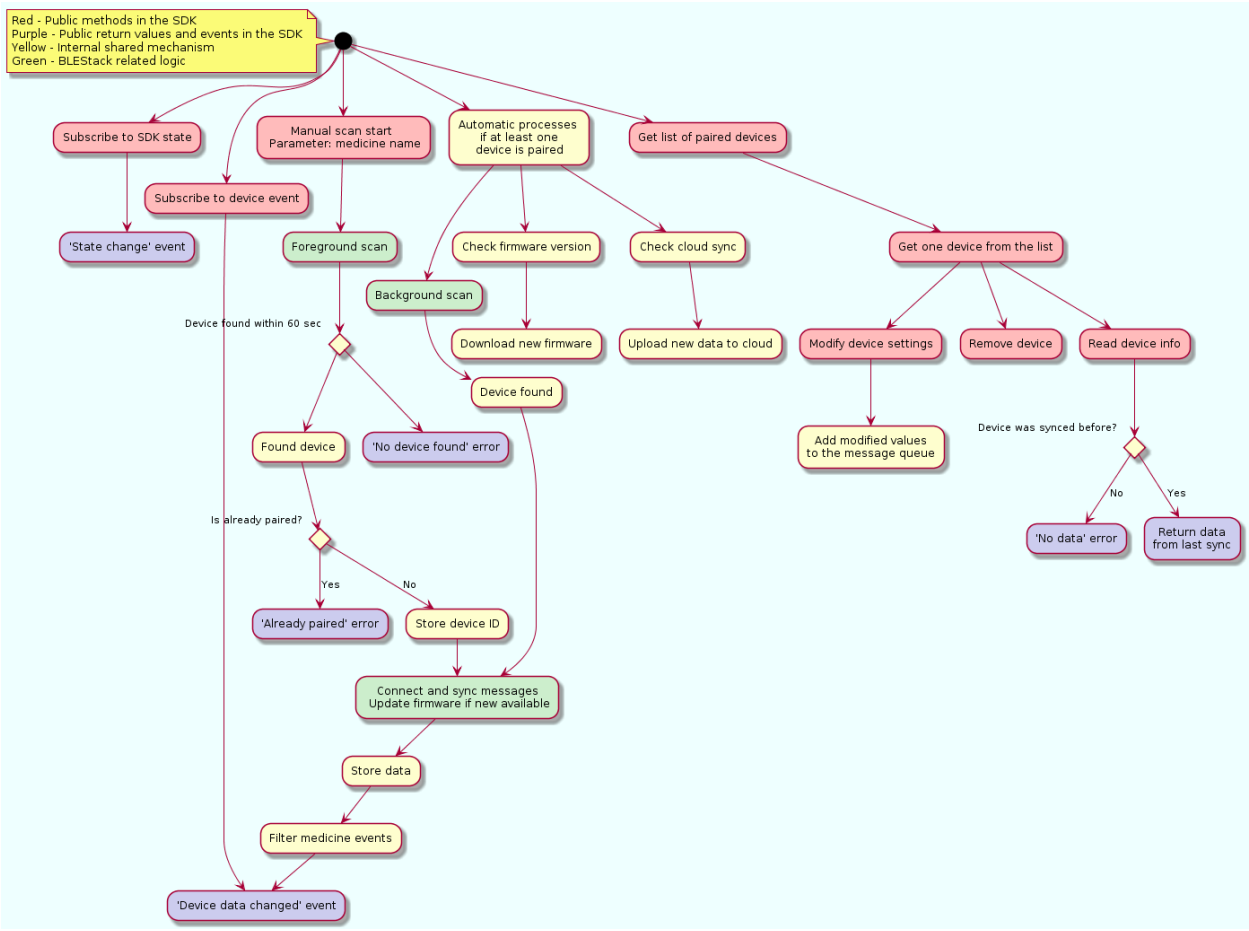

Figure 5: Activity diagram of the SDK

High-level components The component diagram (Figure 4) defines the multi-platform setup of the SDK, through the largest internal and external modules.

Because a significant part of the existing codebase had to be mixed with the shared code, this illustration also helped to document the interconnection between the platform specific and common packages. The hierarchy of the modules on the diagram is similar to the folder structure of the project, even though at this point the actual implementation was not considered. The diagram suggests that the components shared code has to be written in Kotlin, and the platform-specific code will always stay in Java or Swift. Later we will see, that in the implementation only the BluetoothStack code remained in the original language, but the platform specific codes were also made with Kotlin. The language specification was too early when we designed the module diagram, but the logical structure remained true.

The illustration also presents the version control system repositories, which will be used to store the source code of the internal and external parts of the SDK. The SDK deployment was not specified with the requirements. In practice, they can be manually shared with the end users. By using a separate repository, this process can be easily automated with a simple copy script in the build system of the main project.

API and internal mechanism The second is a process diagram (Figure 5), which presents the functionality of the public API along with the main underlying logic. The list of processes is based on the feature set of the existing application, in addition over the public interfaces as the highest layer.

The color coding on the figure separates the internal and external processes: red and purple represent the public interfaces, while green and yellow the internal mechanism. Apart from the automatic background features, every state starts with a red element, which is a function in the SDK that is executed by the host application. The results of these calls are sent back to the application through the events that are described with the purple boxes.

This diagram was utilized to review the specification of the required features and define the visibility boundaries during the implementation.

Design patterns. Based on the public SDK samples and the size of the project, we decided that the best approach for the API design is to bundle the features of the SDK in one public class. This idea matches the application of the Facade design pattern. Our approach not only makes the access to the features easier, but also prevents the host application to access the internal implementation of the SDK.

The implementation of the API Facade class will be described later; however, it is important to note, that the Facade is not the only public component of the architecture. The classes of the incoming and outgoing objects also need to have public visibility. The Facade should be the only part of the public SDK where the host application can execute code.

The list interfaces in the API will also include Observers for the asynchronous events. This includes the broadcasting of the SDK state changes, device events and the result of the pairing.

4.4 Summary

The results of the requirements specification, codebase analysis, and the architectural plan diagrams define the design of the SDK. It is important to note, that the diagrams are only functions as the summary of the features, the real implementation should be based on the requirements and the functionality of the existing software. The next chapter will demonstrate the plan in practice, through the proof of concept implementation of the SDK.

Apart from the selected techniques, there are more approaches to define the architecture with greater details. Due to the relatively small feature set of the system, this was not necessary in our case, but more complex SDKs might require additional steps. Some additional perspectives presented by Rozanski and Woods [15] which were not analysed for the architecture definition: scalability, performance, legal regulations, development resources. Part of these aspects will be analysed after the proof of concept implementation, which makes the evaluation easier in cases like the performance or stability measurements.

Validation. The proof of concept implementation of the SDK works as a skeleton system[15] validation technique: The first half of the implementation will focus on the high-level components, and if the structure is suitable for the selected use-case, the development will continue. The literature also suggests the use of formal reviews and structured walkthroughs[15], which were made through the internal specification meetings with the business case provider as the plan documentation evolved.

The analysis of the proof of concept system and the summary of the development meetings will be discussed in the evaluation chapter.

5 Implementation

Compared to a standard mobile development process, the implementation of the proof of concept SDK presents several extra challenges. Due to the experimental multi-platform setup in Kotlin/Native, the configuration process needs to be discussed in-depth. The API is the core part of the final product, which also deserves a separate subsection for further analysis. Since most of the device management implementation is based on the existing software of the business case provider, their inclusion is presented as a refactoring process in the new environment.

The current chapter includes several listings of the referenced source codes. These examples are not always fully representing their original counterparts, either because the underlying implementation is irrelevant or the source code is a private intellectual property of the company. The omitted parts of the sources are indicated with the "..." notation. Despite the restrictions, the included listings should be sufficient examples to demonstrate the related concept of the discussion.

5.1 Kotlin/Native

5.1.1 Brief overview of Kotlin

Kotlin is a relatively young programming language, first introduced in 2011 by JetBrains. It is a general-purpose language which mainly uses the Java Virtual Machine (JVM) as a build target, and it can be fully interoperated with Java.[9] As Kotlin offers many modern language features and has a cleaner syntax than Java, it became a popular standard in Android development. Kotlin is also supported by Google, as they made it an official language alternative in the Android Studio development environment in 2017.

Kotlin is developed as an open-source project; therefore it benefits from the direct feedback from their target audience. The open-source repository and the core language features are maintained by JetBrains, who also has a commercial interest in the language. This makes the development of Kotlin more dynamic compared to other open-source languages, where there is no direct business interest in the support. However, their influence does not affect the usage of the language, as Kotlin can be used in development without the commercial tools provided by JetBrains.

Kotlin includes many modern features which are present in similar modern languages like Rust, Go, and Swift. It allows both object-oriented and functional programming styles, includes smart casting, higher-order functions, extension functions, non-nullable types and type inference.[19] This allows Kotlin programmers to write shorter and more type-safe code than in Java. Since it is fully interoperable with Java source files, it can be used in existing Android projects without restructuring the sources.

Learning a new programming language is always a challenging task, but compared to Java, Kotlin offers a more ergonomic syntax. Through this modern language, newcomers can learn the basics of Android development more quickly. The case of

[19] https://kotlinlang.org/docs/reference/faq.html

Kotlin is similar to the changes in the iOS development community, where Objective-C, the original development language was replaced by Swift, which became a highly successful alternative. Just four years after the initial release, the number of open-source iOS projects written in Swift has surpassed the ones which were made in Objective-C.[20] A similar tendency is expected for Kotlin.

5.1.2 The use-case of Kotlin/Native

Even though Kotlin was initially developed to target the JVM platform, the language itself can be utilized in different contexts. In 2017 JetBrains introduced the Kotlin/Native project[21], which is a technology that allows the compilation of Kotlin to standalone native binaries with the LLVM compiler.[22] The LLVM compiler provides native support for development on iOS, MacOS, Android, Windows, Linux, and WebAssembly. Kotlin/Native also allows two-way interoperability with existing native sources, including C, C++, Objective-C, and Swift. Native C/C++ libraries and iOS frameworks can also be bridged in a Kotlin/Native project, which allows the extension of the development with the platform-specific tools. This means that a Kotlin source compiled with Kotlin/Native can be included in any existing native iOS project.

Kotlin/Native is already capable of producing stable binaries; however, their usage in multi-platform projects is still an experimental feature. The development of cross-platform software is helped by the kotlin-multiplatform Gradle[23] plugin. This plugin manages the build and deployment processes of multiple target platforms in a Gradle build tool based project. Creating a new cross-platform application with the plugin is a complicated procedure, but fortunately, some open-source sample projects are available for reference. Still, a production-ready setup requires fine-tuning of the build process, which will be discussed later in this chapter.

5.1.3 Platform-specific Kotlin source

The main goal of a Kotlin/Native multi-platform project is to provide a shared codebase between the native targets. Since there are many low-level differences in the standard libraries of the platforms, their usage needs to be bridged in the common implementation. This can be done through the expect/actual mechanism, which is provided by the kotlin-multiplatform plugin.[24] The declaration of an abstract class or variable with an expect modifier tells the build system that an actual implementation is required on each target platform. An actual implementation is included in the source code of each target platform, which is based on the expect counterpart.

One practical example of a platform-specific declaration is the access to the local date, which is also used in the development of the SDK. Since date classes have different interfaces on different systems, the access of the current time is not a trivial

[20] https://www.benfrederickson.com/ranking-programming-languages-by-github-users/
[21] https://github.com/JetBrains/kotlin-native
[22] https://llvm.org/
[23] https://gradle.org/
[24] https://kotlinlang.org/docs/reference/platform-specific-declarations.html

task in the shared code. For simplicity, we have only implemented a timestamp functionality in the common date class. The timestamp is represented as a long data type, and it includes the current Unix time, which is the number of elapsed seconds since January 1, 1970. This concept is available both on Android and iOS, but their access is different in their date representation. First, we had to define the expected static class in our shared code, which is shown on Listing 1. This expected class can be used in the common code after the declaration, and if the actual implementations are missing the build system will give an error.

```
expect object SharedDate {
  fun nowUnixLong(): Long
}
```

Listing 1: Expect declaration of a date class in a shared Kotlin code.

Based on the expected class, we have to create the actual implementations for the iOS and Android targets, which are included on Listing 2 and Listing 3. Note that the iOS implementation uses the NSDate class which is included in the iOS Foundation library that includes the most important data types and collections for the system. This library is automatically interoperated to Kotlin with the help of the kotlin-multiplatform plugin, which allows the usage in the project. The Android implementation uses the Java.util package, which is automatically bridged to Kotlin thanks to the Java interoperability.

```
import platform.Foundation.NSDate
import platform.Foundation.timeIntervalSince1970

actual object SharedDate {
  actual fun nowUnixLong(): Long {
    return NSDate().timeIntervalSince1970.toLong()
  }
}
```

Listing 2: Actual declaration of a date class in the iOS target written in Kotlin.

```
import java.util.*

actual object SharedDate {
  actual fun nowUnixLong(): Long {
        return Date().time
    }
}
```

Listing 3: Actual declaration of a date class in the Android target written in Kotlin.

If a feature cannot be mapped to a shared implementation in the common package, they can also be supplied as a separate source in the output binary. A practical example is the Android application context, which is needed for the database setup. There is no similar concept on the iOS side; therefore the database is configured separately. In general, Kotlin/Native allows the reusability of the sources but does not prevent the development of standalone features in different targets.

5.2 Configuration

The configuration of a Kotlin/Native multi-platform project is challenging for various reasons: The setup of custom configurations requires scripting knowledge in the environment of each target platforms. The amount of Kotlin/Native sample projects are limited, and some of them are even deprecated, due to the breaking API changes during the recent months. Fortunately, the recent changes made the general configuration easier, and the community is still working on improvements for the development setup.

Based on the sample codes, the development of the SDK is configured as a Gradle[25] project. The software can be developed with different IDEs or in a command line. We have selected Android Studio as our development environment, which both supports standalone Gradle builds and the visual debugging of Android applications.

Since the kotlin-multiplatform plugin is still under heavy development, in the first months of 2019 the new version release also made the general setup easier. The recent samples include a blank setup for a simple cross-platform application on iOS and Android. This is an excellent basis for the development, and in most of the cases it is enough for production releases.

In our case, the configuration is slightly more complicated and required additional research for an adequate setup. This includes the bundling of the platform-specific modules and the setup of the critical dependencies and the configuration of the distribution. In some cases, there was no available documentation or source code to help us achieve our goals. To overcome these barriers, the contributors of the related open-source projects also had to be contacted. This section summarizes the challenges and results of the development configuration.

5.2.1 Bundling existing code

A central point in the requirement specification is the inclusion of the BluetoothStack module in the SDK bundle. This process is part of the configuration and has to be managed separately for Android and iOS. Before the discussion of the configuration, two related definition needs to be introduced:

- **Universal binary:** Both iOS and Android-based smartphones include a variety of different CPU architectures. During the development of a standalone application, the debugging of the software is only done with a selected architecture of a target device. Usually, only the release version includes all supported architecture of the application. During an SDK development, we have to ensure that every included dependency contains a universal binary, to support the development of the host application. Moses [13] defines a universal binary as: "Also referred to as a fat binary, a universal binary is a combination of multiple computing architecture instruction sets in a single program or application."

[25] Gradle is a popular open-source development tool, which allows the automation of the software building, dependency management, and releases. Official website: https://gradle.org/

- **Umbrella framework:** Clair [4] introduces the concept with a short definition: "Umbrella frameworks are frameworks that contain two or more other frameworks." Based on our previous notes on the difference between a framework and a library, we can use the umbrella library expression to describe libraries which contains additional libraries. This approach is usually used by developers to bundle third-party dependencies in their framework, which could lead to conflicts in a host application; therefore it is considered as an anti-pattern, and their usage is discouraged by Apple.[26] However, if the umbrella only contains sources from the original developers, the side effects can be prevented, an in some cases it can be a viable solution to bundle components.

Java sources in Kotlin/Native. As Kotlin supports Java interoperability on the language level, mixing existing Java source codes in a Kotlin/Native project is straightforward. The Android version of the BluetoothStack uses Java standard libraries, Android libraries, and third-party dependencies. The Java and Android libraries can be accessed in the Kotlin/Native project if the sources are included in the folder of the Android target. The third-party dependencies are a bit more challenging to manage. By default the kotlin-multiplatform module produces an AAR library, which is a standard format for Android applications.[27] Since the bundling of the third-party dependencies in an umbrella library fashion is discouraged, they need to be included in the host application manually. This means that the integration is slightly more difficult compared to the iOS counterpart, and this should be highlighted in the documentation of the SDK.

Swift sources in Kotlin/Native The integration of existing iOS sources is a non-trivial process. Currently, standalone Swift sources cannot be interoperated to Kotlin/Native, only Objective-C codes.[28] However, because Swift natively supports bridging to Objective-C, the existing Swift files can be used in the SDK. The interoperation process of the language requires the input sources in a library format, which can also be a native iOS framework. For this reason, we have created an iOS subproject in the SDK repository which produces a library that includes the BluetoothStack sources in Swift with Objective-C bridging. Typically, a release version of an iOS framework only contains a binary for the selected architecture in the development environment. For the SDK distribution process, we need to support all target architectures of the host application. To solve this, we have utilized a custom build script which produces a universal binary in the output.[29] With an additional script, it is automatically copied into the SDK source folder, which can be interoperated to Kotlin. The output is named as PlatformLib, which is also the package name when referenced in the Kotlin/Native source.

[26] https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/CreationGuidelines.html

[27] https://developer.android.com/studio/projects/android-library

[28] https://kotlinlang.org/docs/reference/native/objc_interop.html

[29] https://gist.github.com/sundeepgupta/3ad9c6106e2cd9f51c68cf9f475191fa

5.2.2 Dependencies

As suggested by the selected SDK development principles, the number of external dependencies should be minimized to prevent side effects on the host application. Therefore, some of the considered libraries were dropped as they were not critical for the implementation. In this section both the considered and included dependencies are presented, with an explanation of their selection process.

The list of considered dependencies, which were not included in the implementation:

- **Multiplatform Bus**[30]**:** As the implementation handles different asynchronous operation on multiple platforms, an event bus is a reasonable addition for managing these processes. Instead of using an additional dependency for this purpose, we have decided to create a custom lightweight bus implementation, which works on both iOS and Android. The considered module was partially used for reference for our version.

- **Klock**[31]**:** The shared Kotlin code does not include a date class, because of the different implementations of the platforms. Klock is a simple library which includes a consistent multi-platform date class. Eventually, we have not used it, since it provides much more features than needed for the SDK. Instead, the collected data is stored with a Unix timestamp, which can be requested easily from both target systems.

- **Console logging libraries:** There are many popular data logging libraries available for Kotlin/Native; however, they usually only consist of one or two classes, which are relatively easy to implement manually. To reduce the number of dependencies, we have made our own custom cross-platform console logging implementation.

- **Ktor**[32]**:** Ktor is an HTTP server-client framework, which can be used to handle web communication in Kotlin applications. The dependency is currently not used, as it was not necessary for the proof-of-concept implementation. Later on, it can be utilized in the development of the web service communication of the SDK.

The list of included dependencies in the SDK implementation:

- **SQLDelight**[33]**:** SQLDelight is a code generator which simplifies the usage of SQLite[34] databases in Kotlin. It also includes native drivers for bridging the cross-platform usage of the database, which makes it possible to write the common operations in the shared code. This dependency significantly reduces the development time; therefore we have selected it to build our persistence

[30] https://github.com/florent37/Multiplatform-Bus
[31] https://github.com/korlibs/klock
[32] https://github.com/ktorio/ktor
[33] https://github.com/square/sqldelight
[34] https://sqlite.org

layer around it. Since SQLite is a widely used technology for mobile databases, regardless of the current limitations of SQLDelight, the future-proofness of the underlying database is guaranteed.

- **Stately**[35]**:** One known limitation of Kotlin/Native is the poor cross-platform multithreading support. Currently, every static class that has a variable should be marked with the @ThreadLocal annotation to allow mutations. This annotation is supported by the native targets, but not in the shared code. Stately allows the usage of the @ThreadLocal in the common implementation, along with other useful tools for state management. Since the effective thread management is still under development in the Kotlin/Native project, this dependency might be removed in the future.

- **Kotlin Coroutines**[36]**:** Kotlin provides language-level support for coroutines, which helps the management of asynchronous operations in the control flow.[37] This is an especially useful feature in the management of the device synchronization process, where many reading and writing operations need to wait for each other. Using coroutines not only makes the code readable, but also reduces the risk of errors and makes the process easier to debug. To use the coroutines library of Kotlin, a platform-specific bridging library needs to be included in a Kotlin/Native project as a dependency.

Apart from the modules used in the source code, some additional dependencies were added to help the software build process. These are not included in the output bundle of the SDK; therefore they do not have any side effect on the host applications. The list of selected build dependencies:

- **Dokka**[38]**:** A documentation generator plugin, which uses Javadoc style comments from Kotlin source files as an input.

- **Detekt**[39]**:** A static code analysis tool. During the build process Detekt verifies numerous code quality rules and highlights potential issues in the source code.

5.2.3 Open-source collaboration

Working on new technologies is not always a disadvantage. Compared to mature frameworks like React Native or Xamarin, Kotlin/Native is still under heavy development and this development means a continuous improvement on the existing features. Even if the changes in the API made many sample codes obsolete, in exchange the new API is easier to use and provides more features. Some configuration still requires manual setup, but the open-source developer community of Kotlin/Native is available for help, and their input helped us to make the configuration production-ready.

[35] https://github.com/touchlab/Stately
[36] https://github.com/Kotlin/kotlinx.coroutines
[37] https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html
[38] https://github.com/Kotlin/dokka
[39] https://github.com/arturbosch/detekt

During the development, I have contacted the developers of the Kotlin/Native and the SQLDelight module through GitHub issues. These issues are public, and they can be referenced for similar problems and in the development stages of their projects. The following list summarises the discussions of the public issues with their outcomes:

- **Kotlin/Native - Jan 18. 2019**[40]
  Title of the issue: Embed native framework in iOS output.
  Context: The existing low-level platform-specific source code for the iOS target is bundled in a framework. This framework is successfully interoperated to Kotlin, however, by default it needs to be manually included in the SDK release bundle, which makes the delivery slightly more complicated. I have reached out to the developers of Kotlin/Native, to help me include the platform-specific framework in the SDK output framework.
  Resolution: My question was answered in-depth, and this helped me to update the build scripts of the SDK to include the platform-specific sources in the SDK, as an umbrella framework. This not only helps the distribution but also improves the security of the SDK by hiding the public headers of the platform-specific framework. The contributors have extended the Kotlin/Native linker module with an additional flag that allows the definition of the paths of the embedded frameworks. I have shared a compact Gradle script in the original issue, which can be used to automatically copy the embedded framework into the main output bundle after the build process.

- **Kotlin/Native - Feb 26. 2019**[41]
  Title of the issue: Common iOS target and interop reference.
  Context: The default output iOS framework of a Kotlin/Native project only supports one CPU architecture, which is selected before the build process. Currently, modern iOS frameworks should support at least two different architectures: ARM 64 which covers all physical iOS devices that were released since the iPhone 5S phone, and x86-64 which is the architecture of the iOS simulators. Therefore, the SDK should support both for production usage, in the form of a fat framework. The configuration and merging of the builds with two different architecture were not clear, especially in the case when an interoperated library is included in both outputs. I have listed my relevant questions and solution approaches in the form of an issue
  Resolution: Based on some existing sample codes and the guidance of the contributors, we have improved the configuration of the build targets and created a script which generates a universal binary in the framework output. This setup was highlighted by the members of the Kotlin/Native community, as it can potentially solve similar issues in the future.

- **SQLDelight - Jan 16. 2019**[42]

[40] https://github.com/JetBrains/kotlin-native/issues/2555
[41] https://github.com/JetBrains/kotlin-native/issues/2718
[42] https://github.com/square/sqldelight/issues/1164

Title of the issue: iOS Driver usage example.
Context: The original guide and sample code of the SQLDelight plugin was not clear enough for practical usage on the iOS platform. This was mostly caused by the breaking API changes of the library, which was not updated in the documentation before I posted the issue.
Resolution: I have highlighted the unclear points in the documentation. As I presented the problems and tried to resolve the issue by myself, I managed to get the plugin working. The contributors of the project updated the documentation to make the setup process more detailed.

- **SQLDelight - Feb 12. 2019**[43]
  Title of the issue: Modifying and saving rows directly.
  Context: The usage of the SQLDelight library requires the definition of the property modifiers for their code generator, which is hard to maintain in a table with many columns. I have proposed to make the generated objects mutable and make the general property modifications easier.
  Resolution: My proposal was declined, but I have received a tip to reduce the required boilerplate. This can be achieved by using a copy constructor of the generated objects and directly saving the modified data. Another user of the library also utilized this suggestion.

- **SQLDelight - Feb 19. 2019**[44]
  Title of the issue: Issues with comments.
  Context: The selected documentation generator of the SDK requires Javadoc comments in the classes as an input. Since the SQLDelight classes are generated, the comments should be included in the input of the generator. The plugin supports this functionality, however, at the time when I posted the issue it was broken if more than one comment were included in the input.
  Resolution: The issue was investigated and marked as a bug. At the time of the writing of this thesis, it is not yet fixed.

5.2.4 Distribution

The final step of the configuration is the setup of the deployment scripts for the delivery process of the SDK. In most cases, libraries are distributed through online repositories with a version control system. Since this was not part of the specification and the SDK will only be used by a small number of customers, we have selected a semi-manual deployment approach. The SDK build artifacts, documentation and sample code, are included in a private Git version control repository. This repository is shared with the customers, who can include it in their software as a subproject. The new versions are pushed to the repository in the form of a Git commit, which they can manually pull to update the SDK.

We have created this repository next to the internal development repository of the SDK in the same root folder. This way, the deployment scripts can use a relative

[43] https://github.com/square/sqldelight/issues/1216
[44] https://github.com/square/sqldelight/issues/1224

path to move the artifacts from the build folders to the distributed repository. These scripts are included in the Gradle configuration and can be linked with the release build tasks.

5.2.5 Summary

The configuration phase of the SDK took significantly more time compared to a standalone library development project in Android or iOS. This was expected due to the application of a new technology and the complex multi-platform setup. In retrospective, the time spent on the configuration already paid off in the API implementation process, as it was fully developed in the shared component. In the end, the main advantage of the shared codebase will mainly be noticeable in long-term maintenance.

The challenges during the configuration also produced valuable public references for future Kotlin/Native developers. It is expected, that the new versions of the kotlin-multiplatform plugin will include a simplified API and more public examples for library development. This experimental configuration can also serve as a basis for the future feature set; therefore I have published the setup in a version control repository.[45] This package only includes the basic framework development configurations for Kotlin/Native, and does not include any third-party modules or private sources of the business case provider company.

5.3 API

This section presents the developed public API of the SDK. Since the implementation only serves as a proof-of-concept solution, only the core device management features included. The further development of the additional requirements can be based on these examples.

Realization of the architectural plan Listing 4 presents the abstract of the PopitSDK singleton class, which includes the public API functions. For better readability, the function implementation and some comments are omitted. A couple of non-trivial example of the Javadoc style comments are kept for reference. These comments are used by the documentation generator plugin which runs during the build process.

The PopitSDK class includes functions to pair a device, read and store the incoming information, write data to the device, and handle unexpected states. These methods are sufficient to demonstrate the usability of Kotlin/Native and create a basis for a production-ready implementation of the SDK.

Properties and functions. The class includes private properties to store a device manager class instance, observers of the host application and the state of the SDK. The device manager object is initialized with the SDK, and the primary function calls

[45] Skeleton project for multi-platform framework development in Kotlin https://github.com/endanke/kotlin-mpp-framework-skeleton

are handled by it. During the initialization process, the class connects to the internal observers to listen to the state changes and forward it to the host application. With the loggingEnabled property, the internal console logs of the SDK can be enabled, which can be useful for debugging.

The startPairing function is the only asynchronous active device operation, which uses a callback property to present the result of a pairing. The result is either an identifier of a successfully paired device, or an error which indicates the end of the default timeout, or an exception.

The specification of the SDK notes that the paired devices can send both utility information and pill taking events to the application. The DeviceStateListener class includes two methods to listen to either of these events. The utility information is stored in the database of the SDK and can be retrieved with the getInfo command. The pill taking events are only provided through the interface callbacks, which forces the host application to make their abstraction around the pill data storage. Because the SDK is capable of handling multiple devices, all device state changes are including the associated device with the event.

```
object PopitSDK {
    private var deviceManager : DeviceManager?
    private var deviceStateListener: DeviceStateListener?
    private var sdkStateListener: SDKStateListener?
    private var state = SDKState.Blank

    /**
     * Property to enable/disabled logging.
     * Disabled by default to utility messages.
     */
    var loggingEnabled = false

    /**
     * This function initializes the SDK.
     * Cannot start scanning or use any device
     * operations before this function is called.
     */
    fun init()

    fun setDeviceStateListener(deviceStateListener:
    DeviceStateListener)

    fun setSDKStateListener(sdkStateListener:
    SDKStateListener)

    /**
     * Start scanning for a new device with automatic pairing.
     * After 30 seconds of timeout with no result
     * the pairing is stopped.
     * @param medicine The name of medicine
     * associated with the device. Cannot be empty.
     * @param handler A ScanResult handler which
     * will be called when the scan finished.
     */
    fun startPairing(medicine: String, handler: PairingResult)

    fun getDevices(): List<Device>

    fun getInfo(device: Device): DeviceInfo?

    fun remove(device: Device)

    fun setSensitivity(device: Device, level: Int)
}
```

Listing 4: Variables and functions of the API singleton

Facade constraints. To meet the proper application of the facade pattern by Vlissides et al. [22], the PopitSDK class minimizes the additional computation on the interfaces and only acts as a middleware between the internal functions and the host application. The main purpose of the class is to hide the underlying mechanism of the SDK with high-level functions. Apart from this class, no other public methods

available from the other shared classes. The default visibility setting is set to internal for the underlying classes of the SDK. It is always recommended to use the strictest possible visibility settings.

Additional public classes Apart from the PopitSDK singleton, additional public helper classes are also included in the SDK bundle. These classes can be separated into two groups as data wrappers and custom exceptions. The data wrapper classes are generated with SQLDelight, and in most cases, they are directly representing rows from the underlying database. These field of these classes are immutable, and they only serve as containers for the incoming data which is forwarded to the host application. The custom exception classes are defined for error handling, which also includes high-level messages about the specific errors.

5.4 Refactoring

Apart from the implementation of the API, the development of the core features of the SDK was mostly a refactoring process of the existing Android application. Generally, the goal of refactoring is to improve the quality and reduce the complexity of the source code, which makes long-term maintenance more manageable. In this SDK development process, we also had to rework the abstraction and the persistence layer to support two different platforms with one codebase. While the Android application is the primary source of the implementation of the SDK features, the iOS application is also analysed to select the best implementation. During the development of the new abstraction, the code quality of the existing source is also checked and improved where possible.

5.4.1 Separation of layers

The existing applications of the business case provider company were developed in an agile fashion, and parts of the source code were built on the top of prototypes. This is a reasonable approach for a startup; however, it also leads to some conceptual problems in the software architecture. The most noticeable issue is the wrong mixture of software layers, which makes the separation of the device management logic complicated.

A recurring case of this issue is the mixture of business logic in the UI classes. Listing 5 presents a function which is originally included in a fragment class in the Android implementation. Fragments are supposed to control the state of GUI elements, and typically only forwards the interactions to data managing classes. This highlighted function shows the pairing process of a device which was found during scanning. The function changes the state of a progress bar in the view, creates a device information object, writes some information to the database, and displays a Snackbar GUI element to present the result. The information object creation and data manipulation should be managed by a separate class, with a callback on the results of the pairing. This would not only make the device pairing UI-agnostic, but also lower the responsibility of this Fragment, make the code more modular and

easier to maintain. Since the SDK does not include a GUI layer, these functions had to be refactored to work in a programmatic context.

```
private void pairActiveDevice() {
    if (mDevice == null) {
        Log.e("Cannot pair the device, no active device");
        return;
    }
    mProgressBar.setVisibility(View.VISIBLE);

    DeviceInfo info = getDeviceInfo();
    if (info == null) {
        info = new DeviceInfo();
        if (mDevice != null) {
            info.deviceId = mDevice.getAddress();
        }
        info.lastDeviceEventlogDownload = DateTime.now();
        info.id = getDatabase().currentDeviceInfoDao().save(info);
    }

    if (info.id != null) {
        getDatabase().currentMedicineDao().clearDeviceInfos(info.id);
        mCurrentMedicine.deviceInfoId = info.id;
        getDatabase().currentMedicineDao().update(mCurrentMedicine);
        syncDeviceInfo();
        try {
            Snackbar snackbar = ...
            snackbar.show();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else {
        Log.e("Can't pair medicine, no device info");
    }
}
```

Listing 5: Original device pairing function in a GUI class.

The refactored version of the device pairing function hides all data operations in the SDK. The usage of the new function in a GUI context is displayed on Listing 6. This new function not only handles the device pairing process but also initiates the Bluetooth scanning, with a timeout for unsuccessful pairing. This process was initially managed in the fragment, but with this new approach, more than 500 lines of code can be removed from the GUI.

```
PopitSDK.INSTANCE.startPairing(medicineName, new PairingResult() {
    @Override
    public void onPairingResult(@Nullable String deviceID, @Nullable
    PopitSDKError error) {
        mProgressBar.setVisibility(View.VISIBLE);

        if(error == null){
            Log.d("Pairing successful");
            Snackbar snackbar = ...
            snackbar.show();
        } else {
            Log.e("Pairing error: " + error.getMessage());
        }
    }
});
```

Listing 6: Usage of the refactored device pairing function in a GUI class.

5.4.2 Elevating the abstraction

The proper separation of the layers was a helpful step to outline the required business logic of the SDK. The device management process is deeply interconnected with the BluetoothStack module, which handles all low-level communication operation between the phone and the Popit Sense device. Fortunately, the BluetoothStack module is well-separated from the other parts of the software. Regardless of the low-level differences between the two platforms, the provided API of the module is very similar. Since they are included in the SDK in a platform-specific bundle, the shared codebase had to be built on top of the abstraction of the low-level data management. Originally, the device manager classes were initializing the features BluetoothStack, including the continuous background monitoring and callbacks of the device events.

In the new implementation, the SDK initiates the BluetoothStack separately, in a class called DroidBLE and IOSBLE, where BLE stands for Bluetooth Low Energy, which indicates the technology of the device. Additionally, a platform-specific class was also added for the active device scanning, which is called merely as Scanner on both targets. The BluetoothStack handles the device communication events separately from the shared device management. When a device is found, the callbacks of the module sends a notification through the internal bus of the SDK. This bus is connected to the DeviceManager class, which handles the common events in the shared code. Using a bus bridges the low-level API differences since the input of the events are transformed into a common format before they get shared through the internal channels. This way, the new implementation entirely separates the low-level communication layer from the device management.

Listing 7 shows the constructor of the DeviceManager class in the SDK implementation. The scanner object is created from the Scanner class of the active platform, which is received through the expect-actual mechanism of Kotlin/Native. This object initiates the general monitoring of the BluetoothStack. After the scanner initialization, the DeviceManager posts an event through the bus to indicate the

monitoring state of the SDK. The constructor also sets the two listeners for the required bus events. The DeviceReady event is called when a device is connected and ready to receive commands. It automatically triggers the synchronization of the pending messages with a DeviceInfoSyncer object. The DeviceEvent observer tells the manager that a new pill event was received from a paired device. This event is forwarded to the host application, but it is also used to store the incoming data in the web service, when available.

```
1  init {
2      scanner = Scanner()
3      scanner!!.startMonitoring()
4
5      Bus.getDefault().post(Events.StateEvent, StateEvent("Monitoring",
       PopitSDK.SDKState.Monitoring))
6
7      Bus.getDefault().addObserver<DeviceReadyEvent>(this, Events.
       DeviceReady) { value ->
8          syncer = DeviceInfoSyncer(value.deviceId)
9          syncer!!.sync {
10             CloudApiManager.tryMessageSync()
11         }
12     }
13
14     Bus.getDefault().addObserver<DeviceEventMessage>(this, Events.
       DeviceEvent) { message ->
15         CloudApiManager.tryMessageSync()
16     }
17 }
```

Listing 7: Constructor of the DeviceManager class.

5.4.3 Device sync process management

Going one level deeper in the refactoring process, the device synchronization process had to be investigated. This is the core part of the device management, as it includes the information retrieval, configuration writing, and firmware updating processes. The DeviceInfoSyncer class manages the synchronization. Even though the name suggests that it only synchronizes information, it is also responsible for the configuration and firmware writing. The process is executed whenever a device connects to the application, and it runs a series of asynchronous operations.

Code smells. Some of the refactoring decisions were based on a list of code smells, which is are typical anti-patterns that can make the code harder to maintain.[7] The list of smells found in the DeviceInfoSyncer class:

- **Large class:** The file of the class has 489 lines, which makes it harder to get a grasp of the functionality on the first view. Some of the functionality (e.g., data parsing) can be moved in a helper class.
- **Long method:** Some methods are above 80 lines, even though they could be separated into multiple smaller functions. For example, the message reading

and writing is handled in the same function, separated with a large if-else block. By separating the two cases in two functions, the class becomes more readable and modular.

- **Comments:** The class includes several unnecessary comments. For example, the writing and reading block is both highlighted by a static constant (WRITE and READ) and a comment which further explains the obvious use-case of the block. In comparison, some of the more complex functions are lacking comments, which makes them harder to understand.

- **Dead code:** Currently only read and write operations supported during the synchronization. In the future, different commands could be sent through the operation queue, which needs different cases to handle the operations. Since these are not defined yet, the handling is also not necessary; therefore it can be excluded from the new implementation.

Operation flow. The only public function of the class is called sync, which executes the pending synchronization operations of a paired device. This function includes several asynchronous function calls, which are nested in each other with their callbacks. This makes the operation flow hard to modify. Listing 8 illustrates this problem without including the actual function names and implementation.

The device synchronization will include additional operations in the future; therefore this process had to be reworked to be more readable and easier to update. To manage the async operations, we have utilized the Kotlin coroutines library.[46] The concept of coroutines is language-agnostic, but Kotlin supports it natively. Their library includes a wide range of functions to manage asynchronous code, which can be used to execute code in parallel or sequentially. In the new DeviceInfoSyncer implementation, the most used coroutines feature is the suspending functions. A function marked as suspending can execute asynchronous code, while a wrapper coroutine is waiting for the result. We have moved the whole firmware update process into a suspending function, which is awaited in the sync method, to ensure that the other operations can continue after the update. The general reading and writing operations are also suspending, which allows us to execute them sequentially and wait for the end of the synchronization. The refactored sync function is included on Listing 9.

[46] https://kotlinlang.org/docs/reference/coroutines-overview.html

```
public void sync() {
    final DeviceOperation op = new DeviceOperation(mDevice);
    PopitDeviceInfo info = getDeviceInfo();

    if (info == null) { ... }

    if (firmwareUpdateCondition) {
        byte[] fw = getFirmware();
        if (fw != null) {
            writeFirmware({
                @Override
                public void onWrite(boolean success) {
                    if (success) {
                        startUpdate({
                            @Override
                            public void onWrite(boolean success) {
                                if (success) { ... }
                            }
                        });
                    } else { ... }
                }
            });
            ...
            return;
        }
    }

    if (fullSyncCondition) { ... }

    if (downloadCondition) {
        ...
        op.readLogFile({
            @Override
            public void onLogFile(byte[] bytes) {
                if (bytes != null) {
                    ...
                } else { ... }
                syncTime();
            }
        });
    } else {
      ...
    }
}
```

Listing 8: Original device synchronization flow outline.

```
fun sync(done: () -> Unit){
    GlobalScope.launch(ApplicationDispatcher) {
        loadInfo()

        val firmwareReboot = async { updateFirmwareIfNeeded() }
        if (firmwareReboot.await()) {
            // Firmware updated, sync stopped
            return@launch
        }

        downloadLogIfNeeded()
        preloadOperations()

        withContext(ApplicationDispatcher) {
            syncMessageQueue()
        }
        withContext(ApplicationDispatcher) {
            sendRebootIfNeeded()
        }

        Bus.getDefault().post(Events.DeviceInfoUpdated,
            DeviceInfoUpdatedEvent(deviceId, info!!))

        done()
    }
}
```

Listing 9: Refactored device synchronization flow.

5.4.4 Persistence

The original persistence layer of the mobile applications uses platform-specific database modules. For iOS, the selected technology is CoreData[47], and for Android it is Room[48]. Both solutions are an abstraction layer above SQLite, which provides additional features like auto-generated tables, queries, and migration management. The database model is very similar on both platforms. They include the same classes for the device management, and some additional data objects for the medicine, reminder and user storage.

For the implementation of the persistence layer in the SDK, we had multiple options: pure SQLite with manual management, an abstraction layer over SQLite or a non-SQLite based solution, like Realm[49]. To match the functionality of the existing applications, we have decided to use an abstraction layer, in the form of the SQLDelight module. Compared to CoreData and Room, it less automatized, but still easier to manage than a pure SQLite database. As described in the configuration section, the setup of SQLDelight had several smaller issues, since it is a relatively new technology compared to the platform default databases. Fortunately, we have overcome the difficulties of the configuration, and the module made it possible to

[47] https://developer.apple.com/documentation/coredata
[48] https://developer.android.com/topic/libraries/architecture/room
[49] https://realm.io/

define and manage the database from the shared code, which saves time over the long term.

Inconsistent class naming. The inconsistent prefixes in the naming of the original data classes is a recurring error. For example, the device information holder class in the original implementation is named as PopitDeviceInfo, but the medicine data is stored in the Medicine class. The usage of the prefixes is likely a habit that is originated from a developer who was using Objective-C on iOS previously. Namespace declaration was a missing feature from Objective-C; therefore general class names were prefixed with the name of the package or the project to prevent collision.[21] With the introduction of Swift, this requirement became obsolete, as the classes can be referenced through modules. In Kotlin, the classes are included in packages, just like in Java, so there is no need to use manual prefixes in the SDK implementation.

SDK data classes. The general approach in the refactoring of the data classes is to aim for the minimal requirements. By excluding the unnecessary classes and properties, we can keep the SDK lightweight and easier to maintain. Since the main purpose of the SDK is to manage the device-specific features, we can discard the medicine and user-related classes.

The original implementation defines a Device and a DeviceInfo class to store the reference of a device separately from the collected data. These classes are linked with relations, which makes the queries easier, but the migration of the database harder. Based on the source code of the existing applications, there was no use-case where both the Device and DeviceInfo data was requested together. Therefore, the implementation of the SDK does not keep a foreign key between the two classes. The DeviceInfo class can be still linked to Device with the unique ID of the devices, but there are no references between the tables, to make the future migrations easier.

For the functional evaluation of the SDK, only four data classes needed to be defined. The following list describes their use-case and operations:

- **Device:** A lightweight object which represents a paired device with the SDK. It includes the unique identifier of a Popit Sense hardware, with an associated medicine that is supplied during the pairing. Removing a Device object from the database is equal to an unpairing action since the automatic synchronization is based on the list of these devices.

- **DeviceInfo:** This class includes every static incoming information from a paired device, including firmware version and battery level. It can be queried through the API from the host application, even after the synchronization process ended.

- **DeviceEvent:** It stores the incoming ad hoc events from the devices, including the pill usage data. The object is automatically forwarded to the host application, but cannot be requested after the notification of the event. This forces the application to build their own abstraction around the collected data. The DeviceEvent objects are stored in the SDK for analytical purposes.

- **DeviceOperation:** The DeviceOperation class defines the pending communication operations between the SDK and a device. Unlike the other classes, this cannot be accessed from the host application, as it is only used for internal purposes.

Property types. An essential difference between the data models of the iOS and Android implementation is the type of their properties. Some values are stored as a 32-bit integer on iOS, while their counterpart is stored as a 16- or 64-bit integer on Android and vice versa. In most of the cases, there was no clear documentation about the required precision, but this can be identified from the context of the property. The data types of the two platforms are also implemented differently. Hence we decided to use Unix timestamps in the shared implementation. The 32-bit integer type can only hold the timestamps until 2038[50]; therefore the new implementation uses 64-bit types to make the system stable even in the far future. The differences in the property types and the used type in the shared implementation are displayed on Table 2. Note that the iOS implementation uses the NSNumber wrapper in the data model definition, which does not give a constraint on the integer precision.[51] In some case, the application uses short or long integers in the wrapper, which cannot be compared in the class definition.

Room (Java)	CoreData (Swift)	SQLDelight (Kotlin)
DateTime	Date	64-bit integer timestamp
String	String	String
Long	NSNumber	Long
Short	NSNumber	Short
Integer	NSNumber	Long

Table 2: Equivalent property types between the platforms.

5.5 Distribution

The distribution of the SDK is managed through a private version control repository, which hosts the release bundle. The bundle contains the release build of the Android AAR library and the iOS framework, along with the platform-specific sample applications and API documentation which covers both platforms.

5.5.1 API documentation

The API reference is created with the Dokka documentation generator module.[52] Since the API is developed in the shared code, the documentation covers both platforms. Even though the data types are using Kotlin language, they can be easily

[50] http://maul.deepsky.com/~merovech/2038.html
[51] https://developer.apple.com/documentation/foundation/nsnumber
[52] https://github.com/Kotlin/dokka

matched to their counterparts in Java and Swift. The inputs of the documentation are Javadoc style comments, which also allows the reference to sample code snippets. Snippets can be supplied in plain text; therefore we included both Android and iOS usage examples for the non-trivial functions.

Listing 10 demonstrates a comment that is used as an input for the documentation generator. The parameters of the function can be explained individually, and the custom classes are referenced to their separate documentation. A sample code snippet is referenced for the function, which is contained in the PopitSDKSamples class under the samplePairing function.

```
1    /**
2     * Start scanning for a new device with automatic pairing.
3     * @param medicine The name of medicine associated with the device.
     Cannot be empty.
4     * @param handler A ScanResult handler which will be called when
    the scan finishes.
5     * @sample [io.popit.sdk.sample.PopitSDKSamples.samplePairing]
6     */
7    fun startPairing(medicine: String, handler: PairingResult) {
8        Logger.log("Scan started")
9        deviceManager?.startPairing(medicine, handler)
10   }
```

Listing 10: Javadoc style comment for the pairing method.

The output of the generator for the demonstrated function is displayed in Figure 6. The basic configuration of Dokka produces a simple HTML page, which can be included in the release bundle. The developers can access the documentation of the parameter types through the automatically generated hyperlinks. The sample output also includes the code snippets, which presents the usage of the function on Android and iOS.

popitsdk / io.popit.sdk.mobile.api / PopitSDK / startPairing

startPairing

```
fun startPairing(medicine: String, handler: PairingResult): Unit
```

Start scanning for a new device with automatic pairing.

```
///// Android
public class MainActivity extends AppCompatActivity implements PairingResult {

    private void startScan() {
        // Pairing a device requires a medicine name.
        // For this example it can be retrieved from the presented text field.
        String medicineName = ((EditText)findViewById(R.id.medicineNameField)).getText().toString();
        // The pairing result will be handled with an interface
        PopitSDK.INSTANCE.startPairing(medicineName, this);
    }

    // This interface method is called one time when the pairing finishes
    @Override
    public void onPairingResult(String deviceID, PopitSDKError error) { ... }
}

///// iOS
class MainViewController: UIViewController, PairingResult {
    // Pairing example on button press
    @IBAction func startPairingPressed(_ sender: Any) {
        // Pairing a device requires a medicine name.
        // For this example it can be retrieved from the presented text field.
        let medicineName = medicineNameField.text!
        // The pairing result will be handled with an interface
        PopitSDK().startPairing(medicine: medicineName, handler: self)
    }


    // This interface method is called one time when the pairing finishes
    func onPairingResult(deviceID: String?, error: PopitSDKError?) { ... }
}
```

Parameters

`medicine` - The name of medicine associated with the device. Cannot be empty.

`handler` - A ScanResult handler which will be called when the scan finishes.

Figure 6: Generated API function documentation with Dokka, including sample codes for both platforms.

5.5.2 Sample applications

Besides the API documentation, two sample applications were made for the release bundle to demonstrate the SDK on the two target platforms. The functionality of the samples are the same, and the function calls are based on the supplied documentation snippets. The source code is carefully documented and includes an example for all API functions. To maintain simplicity, the samples only display the incoming data without any post-processing.

Figure 7 demonstrates the iOS version of the sample application, which has a similar user interface like the Android counterpart. The main screen includes a text input field for the medicine name, a button to start the pairing of the device, and a link to a different screen which shows the incoming messages. The screen also indicates the status of the SDK and the list of the paired devices.

After supplying a medicine name and pressing the pairing button, the software automatically connects to a nearby Popit Sense device, if the hardware is in pairing mode. The second state of the figure shows a paired device, which also displays the hardware identifier, firmware version, and the battery level percentage.

The third state displays the incoming messages screen, which includes the pill taking events from the device. When the developers restart the sample application, the list of paired devices are still available, but the incoming messages are cleared. As described in the integration guide, these messages should be stored by the host application for long-term medicine tracking.

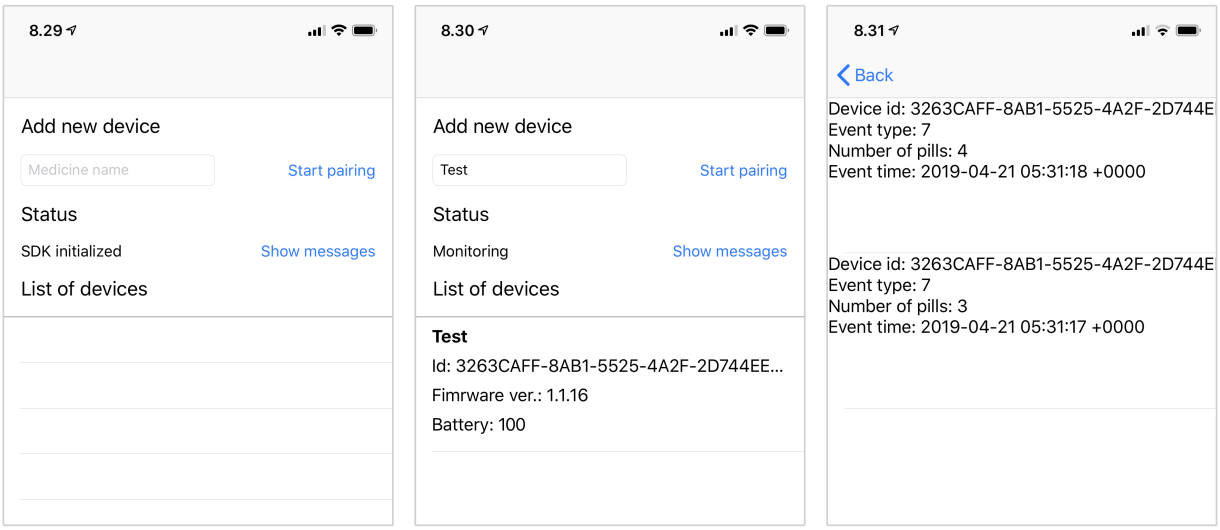

Figure 7: Three states of the sample application: idle, device paired, incoming events.

6 Evaluation and discussion

6.1 Meeting the requirements

The first part of the evaluation is set to collect the necessary data for answering research question 1. As noted in the requirements specification, due to the time limitation of the thesis work, the SDK was only developed as a proof of concept prototype for the selected technologies and practices. The development focused on the implementation of the core features, which are enough to verify every layer of the architecture, the deployment, and the production application of the SDK. The first research question is related to the non-functional requirements of the SDK, with a particular focus on three selected properties: security, easy integration, and future-proofness. This section presents the different approaches in their verification, including both qualitative and quantitative methods.

6.1.1 Functional requirements

Before discussing the properties of the research question, the functional requirements should be evaluated, as the non-functional points assume a correct functionality.

The implementation of the SDK includes all necessary methods to pair a device, conduct two-way communication between the SDK and the hardware, and automatically store the incoming data. The firmware update process and the web service communication features were omitted, as they are not required to verify the architecture and the selected technology. Apart from the custom device configurations, all public interfaces have been implemented, which are used during the demonstration of the API. Figure 8 presents the activity diagram of the architecture, excluding the not implemented features. The SDK includes all high-level modules, build configuration and deployment scripts which are necessary for the evaluation.

Software testing. The SDK was tested with a black box approach through the public API. First, the expected behavior of the classes and functions were tested with the developed sample applications. Since most of the features require a paired Popit Sense device, the testing required manual input. After every major modification in the SDK, the features were tested in the sample applications on both platforms based on the following steps:

1. Clean application install.
2. Pair a Popit Sense device, check if the SDK state is displayed correctly in the sample application.
3. Wait for first incoming information, including the battery level and serial number which are displayed in the application.
4. Take a pill with the device, check the incoming pill count in the application.
5. Restart the application and check if the information is correctly persisted.

6. Take a pill with the device while the application is in the background. Check the incoming pill count.

7. Remove the device from the application.

If all of these steps are successful, the manual test can be considered correct. During the development, the incorrect user inputs were not tested since the current state of the SDK is only used to verify the architecture and the technology. At this stage, manual testing was enough to demonstrate the implemented functional requirements. In the future, this process could be automated with a device simulator, which executes the user interactions based on the test cases.

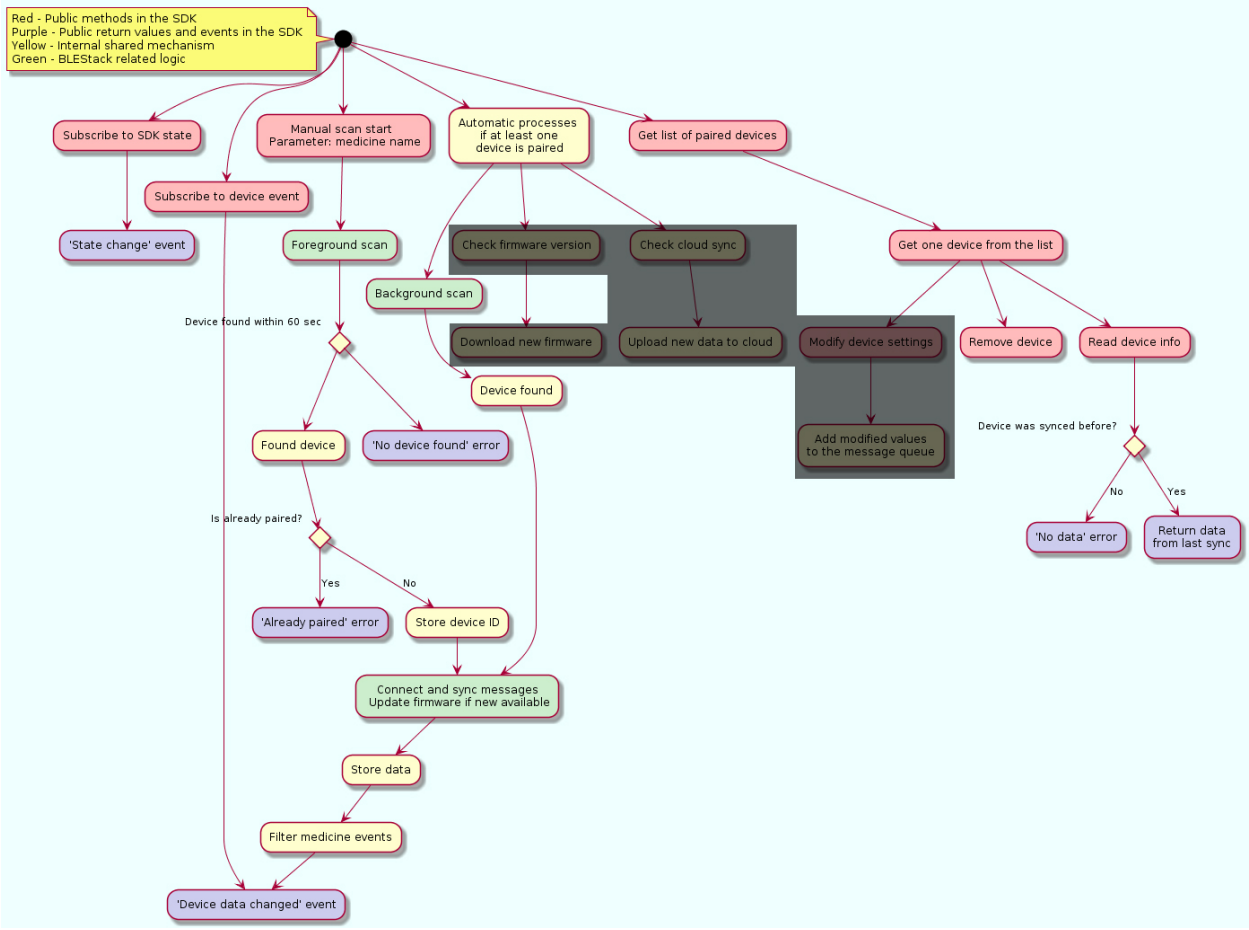

Figure 8: The implemented activities of the proof of concept SDK. The dark boxes hides the excluded features.

6.1.2 Interviews

It is necessary to carry out interviews for the unbiased assessment of the non-functional requirements. The evaluation included various interviews with experienced developers who provided feedback about the architectural design and development decision.

Since the SDK is a private intellectual property of the business case provider company, the interviews were separated into two groups based on the access to the software sources:

1. **Authorized:** Includes the developers within the company, authorized external developers, and the customer companies who are allowed to access the SDK artifacts, sample codes, and documentation.

2. **Unauthorized:** The business case provider agreed that some non-sensitive information could be shared about the SDK, which includes the development configuration in Kotlin/Native and high-level architecture. These are evaluated with the help of senior software developers.

Because the participants had a different relationship with the SDK, the interviews were conducted at different phases of the development, which guided the evolution of the software. All of these interviews were planned as an informal discussion about one selected topic. The following list presents the interviews in chronological order, with the context and the summarized results.

1. **Phase:** Specification.
   Participant: Authorized senior developer.
   Subject: Evaluation of the design decisions and selected technology.
   Summary: The first consultation about the SDK was scheduled as the final step of the specification phase. I have presented the requirements, architectural diagrams, and the selected technology for a senior software engineer, who was authorized by the company to review the decisions.

   The architectural plan was not criticized, as he noted that the proof of concept implementation would uncover the potential issues. He expressed his concerns about Kotlin/Native, which might not be mature enough to build a production-ready software in it. He also noted that his preferred approach in multi-platform SDK development is either to use C/C++ or standalone platform-specific implementations. Based on his experience, it is hard to find mobile developers who are willing to maintain C/C++ sources; thus for small scale projects, he recommends the development in two separate native projects for Android and iOS. We agreed that Kotlin/Native in the worst case could cover the Android platform; therefore it is a good decision to experiment with the technology, even if it cannot meet the standards on iOS.

   Additionally, we discussed the distribution of the SDK artifacts, which was helpful guidance for implementing the configuration. He believes that developers prefer sample applications over detailed API documentation, therefore I should focus more on the development of clear examples. Finally, for my background study of the configuration, I have asked if it is possible to hide the dependencies of the Android library with an umbrella framework approach. He explained that this is a bad practice, as it could lead to conflicts between the dependencies of the host application, so it is better to describe the dependency configuration in the documentation.

2. **Phase:** Configuration.
   Participant: Open-source contributors of Kotlin.
   Subject: Guidance on the proper configuration of the framework development setup.
   Summary: Although the discussions of the configuration in the open-source community might not be considered as a classical interview, it was an important part of the evaluation of the SDK. These discussions are only related to the functional requirements of the implementation; however, they indirectly provide feedback on research question 2. The summary of the discussions were presented in the implementation chapter.

   The main takeaway from the discussions is that some of the required configurations of a production-ready framework development were not part of the kotlin-multiplatform plugin. After the public discussion of the correct setup, more than one person has expressed their interest in the solution on the open-source forums.

3. **Phase:** Development.
   Participant: Authorized senior developer.
   Subject: Evaluation of security decisions, documentation, and API.
   Summary: During the implementation of the SDK the company arranged an additional meeting with the senior software engineer, who provided feedback on the configuration. The second time the discussion focused on the evaluation of the API and the security considerations.

   The feedback on the public functions and classes were positive. The participant stated that the API is easy to understand, mostly because it does not contain too many features. At this point, the asynchronous public observers of the SDK were only partially implemented, and not supported more than one listeners on one observer endpoint. He noted, that multiple listeners might not be needed, as the SDK is mostly only used in the root of the application and will not be reused in different contexts. Additionally, he suggested, that the documentation should include source code snippets for the public functions, which would make it easier to understand the functionality. However, this does not imply that sample applications are not needed.

   From the security perspective, we have discussed the risks of the reverse engineering. He noted, that this might not be an issue if the source code does not contain any sensitive information. The reverse engineered source code of the files are obfuscated; therefore even if someone learns the internal functionalities, it is still hard to reproduce a standalone copy of the SDK. Logically, the development cost of a standalone SDK from reverse engineering would be higher than purchasing the license. The encryption of the internal database of the SDK can be justified with the same reasons. He added, that it might be a good idea to use API keys which are generated uniquely for customers, which could help us to identify the end-users of the product. This would ensure that the SDK cannot be reused outside of the allowed scope.

We have also reviewed the error handling of the SDK. Since the original source code was not error-free, the SDK might also have some unknown built-in errors. He suggested that these errors should be investigated and handled internally. Usually, it would be a bad practice to handle every unexpected error in the root of the SDK, but until we find all unexpected issues, it is better to catch all exceptions than letting them crash the host application.

The summary of the interview also suggested, that there is a need for a list of changes in the SDK features. At this point, the API was only partially implemented, and for the preparation of a future interview, it is useful to know the differences in the functionality.

4. **Phase:** Development.
   Participant: A customer of the business case provider company.
   Subject: Evaluation of the public API, sample codes and integration process.
   Summary: To finalize the proof of concept implementation of the SDK, the business case provider company has arranged a meeting with one of their potential customers who will utilize the SDK. During the meeting, we have presented the API, documentation, and the sample codes, which were discussed to reveal the unclear parts of the integration process.

   The main issue with the documentation was the lack of specification for the hardware component. Most of the questions of the developers were related to the functionality of the Popit Sense device. The hardware documentation was not part of the original specification, partially because the company does not want to share too many details about their intellectual property. However, the discussion has shown that the SDK cannot be used without a high-level overview of the hardware.

   Thanks to the shared modules, the demonstration of the API functions interfaces covered both platforms at the same time, which made the presentation shorter. The developers have noted that the usage is clear, and they can make the integration of the SDK based on the supplied sample applications. Android applications require opt-in location permission for Bluetooth scanning, which caused a minor confusion during about the integration. It is not possible currently to provide this functionality without requesting the permission.

   Additionally, the developers have mentioned that a simulator of the hardware would be useful for their testing processes.

5. **Phase:** Evaluation.
   Participant: Developers of the business case provider company.
   Subject: Handover of the sources, evaluation of the development documentation.
   Summary: So far, the previous discussions have focused on the integration and the security. The third main non-functional requirement is the future-proofness, which partially depends on the maintainability of the sources. The proof of concept implementation will be further developed by the business case

provider company to make a full-scale product from it. Therefore, the project was handed over to the developers within the company, and this process was documented as an additional interview.

To make the handover progress easier, I have created a developer documentation with the summary of the configuration, build tasks, and the main caveats of the implementation. The discussion of this document suggests that the Kotlin/Native configuration has a steep learning curve, especially for the developers who are not experienced in Android development. For testing the development environment setup and the understanding of the documentation, the developers had to implement basic features in the SDK, which are included in both the Android and iOS targets.

The number of notes about the caveats in the development documentation also suggests that the system is hard to manage without enough experience in Kotlin/Native. Most of the issues are related to the SQLDelight module, which will be fixed in future releases. Some configurations could be simplified by linking them to the build tasks. For example, the PlatformLib framework interoperation configuration had to be generated manually before running the iOS builds. This task was connected to the build process after the discussions, to make the future development more straightforward.

One of the developers has noticed an additional issue in the setup, if the SDK is placed in a folder which includes spaces in the path name. This was caused by an incomplete configuration, which can be fixed in a later update.

6. **Phase:** Evaluation.
   Participant: Unauthorized senior developer.
   Subject: High-level design decisions, development process.
   Summary: One additional interview was carried out with a senior software engineer, who is specialized in iOS development. Since he was not authorized to access the internal implementation of the SDK, we only discussed the general SDK development process in Kotlin/Native. The interview started with an introduction to the selected technology and the framework development configuration. He has highlighted the expect-actual mechanism of Kotlin/Native as a useful solution for cross-platform code sharing and further verified that this language is easier to maintain than C/C++.

   He also added, that the recent introduction of Swift 5 brought application binary interface (ABI) stability to the language, which is an important addition to our setup.[53] ABI stability makes the platform-specific framework bundling future-proof, as it ensures that the Swift 5 binaries compatible with the future versions of the compiler.

   Further on, we discussed the testing setup of the SDK, where he also noted that an emulator would be an essential addition to automate the testing. This

[53] https://swift.org/blog/abi-stability-and-more/

would not only make the SDK more stable but can potentially save money in the long run, by eliminating the costly manual testing.

Additionally, we had a conversation about the fat framework bundles, which are generated for the iOS output. He noted that the publication process should be investigated, since a limitation of the Apple App Store might trigger a rejection if the framework contains the iOS simulator architecture. This might not be a problem in the future, but the problem currently can be easily solved with an additional build script that removes the unused targets from the package.

6.1.3 Security evaluation

The OWASP Mobile Application Security Checklist[54] is an industry standard guideline for mobile security, which was reviewed for the evaluation of the SDK. The security checklist presents a set of common vulnerabilities that are collected by Schleier et al. [17]. Even though it was designed for full-scale mobile applications, most of the points are also relevant for SDKs. By analysing our solution through the points of the list, we can rule out the well-known security issues of the software.

The checklist separately categorizes the points for the mobile platforms and general or reverse engineering related vulnerabilities. The list of the general security requirements includes both architectural, network communication, and even code quality related issues. For the evaluation, we have used version 1.1.0 of the checklist.

The points of the list are separated based on their Mobile Application Security Verification Standard (MASVS) level.[55] Level 1 covers standard security procedures, level 2 is more in-depth, and the third layer of defense is reverse engineering protection. We have considered all three levels, as the SDK handles data for a medical device. Even though most of the points are not applicable, since the SDK is not a standalone application and currently does not contain any network operations. The points of the lists are either marked with a pass, fail or not applicable, and explained with a relevant testing step.

For privacy reasons, the full evaluation of the checklist is not included in the thesis. However, those points which have uncovered security shortcomings are included in the following list:

- **1.9 A mechanism for enforcing updates of the mobile app exists.** Currently, the SDK can only be updated manually in the host applications; however unexpected security issues might require a forced update in the future. A temporary workaround could be a remote kill switch, that could disable the functionality of the SDK in case of an unexpected security breach.

- **2.12 The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.**

[54] https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide
[55] https://github.com/OWASP/owasp-masvs

The SDK documentation does not contain a summary of the collected information nor the scope of the data process. The final version should include this, which has to be presented through the host application.

- **6.2 All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.**
  So far the only free text input of the API is the name of the associated medicine for a device pairing. This field should be tested for vulnerabilities and sanitized adequately before the execution.

- **7.5 All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.**
  Before the production release, the included dependencies should also be tested for vulnerabilities. Fortunately, the number of dependencies are kept minimal, and all of the included libraries are open-source, which makes the testing more accessible.

- **8.1 The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.**
  As noted during the interviews, the prevention of reverse engineering is a costly process and might not be too crucial in our case. However, the reverse engineering process can be made more complicated if the SDK can detect devices with root access and block the data management functionalities.

- **8.5 The app detects, and responds to, being run in an emulator.**
  Similarly to the previous point, the release build could check the environment of the execution and block some of the functionalities if the SDK is used in an emulator. This could further complicate the reverse engineering process.

- **8.10 The app implements a 'device binding' functionality using a device fingerprint derived from multiple properties unique to the device.**
  Apart from implementing a customer-specific API key, the devices of the end-users could be tracked with anonymous fingerprints. This could prevent the portability of the application outside from the regulated environment, and reduce the chance of reverse engineering.

6.2 Development with Kotlin/Native

Based on the developed SDK, the selected technology can be evaluated to answer research question 2. During the implementation the positive and negative traits of Kotlin/Native were gathered, which are summarized on the following two lists:

6.2.1 Challenges

- **Complex setup.** Currently, the main issue of the multi-platform programming in Kotlin/Native is the configuration. The technology has a steep learning curve, where the developers need to understand the build process in-depth, and manually configure the project to support the SDK development. Fortunately, the developers of Kotlin/Native are working on new features which will reduce the need for custom scripts, but until then the long-term maintenance is a difficult task.

- **Breaking API changes.** The improvements of the kotlin-multiplatform module also causes breaking API changes, which makes it harder to update the existing configuration. To date, Kotlin/Native had thirty major releases; however, the kotlin-multiplatform plugin was only introduced in July 2018.[56] It is expected, that the plugin will introduce more breaking changes in the coming year. The future stability of the plugin might also improve the popularity of the technology.

- **Small amount of examples.** Due to the experimental nature, only a small amount of open-source projects were made in Kotlin/Native. The breaking API changes also limit the useful examples, as most of the public samples get deprecated if their contributors do not update them. The small amount of examples makes the configuration harder, and the harder configuration means fewer users of the technology who could produce the examples.

- **Small amount of third-party modules.** Even though Kotlin is actively used in Android development during the recent years, most of the third-party modules written in Kotlin cannot be used in Kotlin/Native. Those modules which reference any Java or Android libraries cannot be used in shared code, which only accepts pure Kotlin sources. Most of these cases the platform-specific libraries could be separated, which makes the current Android solutions a basis for future modules.

- **Unexpected problems.** The immaturity of the technology currently causes several smaller issues, which might even lead to unexpected security vulnerabilities. A practical example is the limited concurrency management capabilities in the shared code, which are currently mitigated with a third-party module.[57] Even though the implementation of the SDK was successful, it is important to keep in mind the possible limitations in the current state of the technology.

6.2.2 Highlights

- **Community support.** The breaking API changes and the limited examples compensated by the support of the open-source community of Kotlin. With the help of the contributors of the Kotlin/Native and SQLDelight projects, I

[56] https://github.com/JetBrains/kotlin-native/blob/master/CHANGELOG.md
[57] https://github.com/touchlab/Stately

was able to configure the SDK development process and overcome most of the unexpected challenges.

- **Effective cross-platform development.** Developing low-level shared code between multiple platforms is a complex process regardless of the selected technology. However, after the complicated setup phase, the development of the business logic was straightforward. Every shared source that has been tested in the Android application was also working on the iOS side. This ensures that the end-users will receive the same experience and spares effort on the development and the testing.

- **Language advantage.** Compared to the leading alternative, Kotlin provides a much modern syntax than C/C++. Most of the Android developers are already familiar with Kotlin, but not necessarily with C/C++. Using Kotlin/Native also removes the need for using three different languages for cross-platform module development.

- **Rapid evolution.** The rapid evolution of technology not only causes breaking changes, but also continuous improvements on the API and the supported features. During the development of the SDK a new release of Kotlin/Native introduced bitcode support, which allows the further optimization of iOS binaries.[58] This is just one example from the dozens of feature requests that have been fulfilled by the contributors during the recent months.

6.3 Best practices evaluation

The thesis work started with a literature study to find the best practice recommendations for research question 3. These resulted in a list of general principles, which were applied during the specification and implementation stage. The following list presents the evaluation of the selected guidelines, based on their utilization:

1. **Aim for simplicity.** Simplicity was a critical factor in the development of the API, which made the integration process more manageable. Even though it is hard to maintain internal simplicity in a cross-platform software, it is an essential rule for the architectural design.

2. **Use minimal external dependencies.** As suggested by multiple sources, we tried to keep the number of dependencies small during the implementation of the SDK. Most of the unexpected issues were coming from the third-party database library, which was eventually resolved, but also demonstrated the problems of the dependencies.

3. **Minimize resource usage.** During the refactoring process of the existing codebase, the data persistence layer was built based on this principle. It is important to keep in mind, that the host application will have a separate set

[58] https://developer.apple.com/library/archive/technotes/tn2151/_index.html

of resources, and the SDK should only store the most important data related to the use-case.

4. **Minimize permissions.** One of the main questions during the presentation of the API was about the permissions of the SDK. The fewer permissions we request, the easier it is to integrate the product and keep the application secure.

5. **Keep security in mind.** The development process showed that it is not only important to secure the internal features of the SDK, but also to protect the distributed binary of the software.

6. **Create sample codes and documentation.** Both the sample codes and the documentation are crucial parts in the ergonomy of the integration. Hardware documentation can be a useful addition to the later iterations of the bundle.

7. **Design for long-term.** Future-proofness was one of the selected requirements of the system. As a medical research can last for years, the long-term compatibility of the SDK is a must-have trait in our case.

8. **Get feedback from developers.** The interviews have helped to shape the development of the system and suggested possible improvements which resulted in higher quality implementation.

6.4 Answering the research questions

Question 1: Does the architectural plan fulfill the three requirements of the SDK? The three main requirements of the SDK are the following selected non-functional traits: secure, easy to integrate, and future-proof. Based on the security evaluation and the feedback from the senior developers, the plan is capable of delivering a secure SDK with some minor modifications. The feedback from the customer company shows that the documentation is satisfactory and the API is easy to integrate.

The degree to which the plan is future-proof is questionable, as the maintainability of the Kotlin/Native setup caused unexpected challenges in the company. However, the selected language and framework development techniques are expected to be supported by the platforms over the long-term. In the worst case scenario, the Kotlin/Native setup can be abandoned, and the developed sources can be reused as a standalone implementation on Android.

With these constraints the business case provider company has accepted the implementation; therefore the plan fulfills the main requirements.

Question 2: Is the Kotlin/Native technology stable enough for enterprise applications? Multi-platform programming in Kotlin/Native is still an experimental feature. Based on the list of challenges and highlights, the technology currently has more shortcomings than advantages. Even though the proof of concept implementation was successful, there might be some additional underlying issues

with the configuration that could cause problems over the long term. Currently, C/C++ is still a better alternative if high stability is a key requirement. Nonetheless, Kotlin/Native is very promising, and it will unquestionably be a standard technology for multi-platform business logic sharing.

In summary, Kotlin/Native not yet stable enough for enterprise applications. This most likely will change by the end of 2019, with new releases of Kotlin.

Question 3: What are the best state-of-the-art SDK development methods? The main focus of this thesis work was to collect the best practices of an SDK development process. The collected principles in the literature study were successfully applied, and they significantly improved the result of the implementation. The list of the principles were:

1. Aim for simplicity.
2. Use minimal external dependencies.
3. Minimize resource usage.
4. Minimize permissions.
5. Keep security in mind.
6. Create sample codes and documentation.
7. Design for the long-term.
8. Get feedback from developers.

The selected technology for an SDK development can depend on different factors. We have seen that the development of an SDK can be done with platform-specific native languages, C/C++, Go Mobile, or Kotlin/Native. Currently, C/C++ is still the best option for cross-platform business logic sharing. Kotlin/Native seems promising and might replace C/C++ in the near future.

7 Summary

7.1 Conclusion

Theoretical results. The original goal of the thesis work included the exploration of best practices and the verification of the Kotlin/Native technology through a business case. Even though the literature study focused on general guidelines, the implementation process provides insight into the practical development of an SDK. Most importantly, the refactoring of two separate applications into a shared codebase is a process that is barely covered by the literature. This thesis work can serve as a software architectural design reference for similar use-cases.

Based on the list of the selected principles, the implementation, and the evaluation, we can conclude the three main quality indicators of an SDK development process:

1. **Simplicity.** The design of the SDK and the API should be as simple as possible. This includes the number of public functions, the ergonomy of the API, resource usage, permission requirements, and the number of internal components. Keeping every aspect simple makes the SDK lightweight, easier to integrate, and easier to maintain.

2. **Context awareness.** The SDK should respect the host application. We should keep in mind during all development phase, that the SDK will always be used as a software component. This is especially important from the perspective of application security and resource management.

3. **Communication.** An SDK is, by definition, used by developers. Therefore, it is necessary to request feedback during the design and implementation process.

Practical results. Apart from the analysis of development techniques, an additional goal of the thesis was the evaluation of the Kotlin/Native technology for an SDK implementation. Unfortunately, the current state of the technology is still very unstable for large scale usage, but it is a promising alternative. Aside from the critical evaluation of the research questions, Kotlin/Native might be already a good choice for several business cases, if the development team is competent enough to keep up with the breaking API changes. The configuration of the kotlin-multiplatform plugin will eventually be easier, which could potentially make Kotlin/Native a better alternative than C/C++ for cross-platform code sharing.

As a practical result of the thesis work, a general configuration for framework development in Kotlin/Native has been published.[59] Based on feedback from the open-source community, several developers are experimenting with a similar setup, which makes the public samples a useful reference in their work.

[59] https://github.com/endanke/kotlin-mpp-framework-skeleton

7.2 Further development

The business case provider company has successfully used the SDK for demonstration. Even though the original goal of the thesis was only to verify the architecture with a partial implementation, the final result includes most of the features of the specification. The project will be carried on with the developers of the company, who may or may not continue with the Kotlin/Native technology. Since the Kotlin sources can be reused in a standalone Android implementation of the SDK, the developers might decide to maintain two separate projects for the two target platforms instead of a cross-platform approach.

From the functionality perspective, the firmware update process and web service communication still need to be implemented. These can be based on the working examples of the device management process. The recommended module for the development of web communication is Ktor[60], which provides both a server and client implementation in Kotlin.

Production release and automated testing. If development continues with the use of Kotlin/Native, the stability of the system needs to be tested more in-depth. The internal testing of the SDK was successful; however, this only included a small number of target devices. A suggested testing strategy would be the integration of the SDK into their existing mobile applications. The updated applications could be rolled out in multiple stages, and the stability of the new system could be measured with a limited user base.

Based on the interviews and the internal testing procedures, a simulator for the Popit Sense device is a necessary component for long-term maintainability. By eliminating the manual tests, the company can save both time and money. A potential implementation could be a testing station, which programmatically executes the manual inputs on a Popit Sense device through physical switches. This ensures that Bluetooth latency is factored in during the testing.

7.3 Future of Kotlin/Native

Overall, the experience of the practical application of Kotlin/Native was very positive. It is convenient to use a modern language for multi-platform programming which requires less code than Java or C/C++ to achieve the same functionality. The open-source community around the technology is very active; therefore the technology is evolving rapidly. It is expected that the multi-platform setup in Kotlin/Native will reach a production-ready maturity level within the coming year.

Third-party modules. The state of the third-party modules is slightly more concerning. Public open-source modules are frequently shared on so-called "awesome lists", which present hand-picked modules for a technology or a language. Even though there are more than 100 modules listed for Kotlin, currently less than 10

[60] https://ktor.io/

modules are listed in a similar collection for Kotlin/Native.[61][62] Fortunately, many Kotlin modules can be converted to Kotlin/Native with a slight change in their abstraction by separating the Android and Java standard libraries from the business logic. It is likely that in the future many more libraries will support multi-platform programming in Kotlin by default.

Long-term expectations. The explicit goal of the developers of Kotlin is to support a wide range of platforms and development techniques. Currently, full-scale application development is only possible on Android, as the Kotlin/Native development setup has minimal support for GUI programming. Likely the future multi-platform development in Kotlin/Native will still focus on business logic sharing, instead of the development of visual components. There are many popular high-level cross-platform development tools available, like React Native or Flutter, which might utilize the low-level support of Kotlin in the future.

Based on the use-case of C/C++ in mobile development, it might be possible that Kotlin will be utilized in compute-intensive software. There are already some public experiments available which demonstrate the low-level functionalities of Kotlin in complex programs, like computer graphics or deep learning.[63][64] The development of high-performing software in Kotlin could revolutionize multi-platform programming in the coming years.

[61] https://github.com/KotlinBy/awesome-kotlin
[62] https://github.com/bipinvaylu/awesome-kotlin-native
[63] https://github.com/kotlin-graphics/gln
[64] https://github.com/sekwiatkowski/komputation

8 Bibliography

References

[1] Apple Inc. Overview of dynamic libraries, 2012. https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/OverviewOfDynamicLibraries.html [Accessed: 24 Mar 2019.].

[2] Apple Inc. What are frameworks?, 2013. https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPFrameworks/Concepts/WhatAreFrameworks.html#//apple_ref/doc/uid/20002303-BBCEIJFI [Accessed: 24 Mar 2019.].

[3] William H. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998. ISBN 0471197130, 9780471197133.

[4] R. Clair. *Learning Objective-C 2.0: A Hands-on Guide to Objective-C for Mac and iOS Developers*. Learning. Pearson Education, 2012. ISBN 9780133047387. URL https://books.google.fi/books?id=KOoS3nIxU-AC.

[5] Alan M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995. ISBN 0-07-015840-1.

[6] Chris Eidhof, Matt Gallagher, and Florian Kugler. App architecture, 2018.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0-201-48567-2.

[8] Google LLC. Create an android library, 2019. https://developer.android.com/studio/projects/android-library [Accessed: 24 Mar 2019.].

[9] David Greenhalgh and Josh Skeen. *Kotlin Programming*. Big Nerd Ranch Guides, 2018. ISBN 9780135165188.

[10] Qusay Hassan. *Internet of Things A to Z: Technologies and Applications*. Wiley-IEEE Press, 06 2018. ISBN 978-1-119-45674-2.

[11] Gal Levinsky. 10 tips on how to build the perfect sdk, 2016. https://dzone.com/articles/10-tips-on-how-to-build-the-perfect-sdk [Accessed: 25 Mar 2019.].

[12] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. *ACM SIGPLAN Notices*, 33, 05 2000. doi: 10.1145/286942.286945.

[13] L. Moses. *Iphone Applications Tune-Up*. Packt Publishing, 2011. ISBN 9781849690355. URL https://books.google.fi/books?id=zQR6UCq4n7IC.

[14] Robosoft Technologies. Best practices in mobile sdk development, 2016. https://medium.com/@Robosoft/cb6316a14406 [Accessed: 25 Mar 2019.].

[15] Nick Rozanski and Eóin Woods. *Software Systems Architecture*. Addison-Wesley Professional, 2005. ISBN 0321112296.

[16] Kristopher Sandoval. What is the difference between an api and an sdk?, 2016. https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/ [Accessed: 18 Feb 2019.].

[17] Sven Schleier, Bernhard Mueller, and Jeroen Willemsen. Owasp mobile security testing guide, 2019. https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide [Accessed: 3 Apr 2019.].

[18] Mike Smales. Best practices for building a world class mobile sdk, 2018. https://medium.com/@mikesmales/b090d9e4b774 [Accessed: 25 Mar 2019.].

[19] Ali Soueidan. The difference between library and framework, 2019. https://alisoueidan.com/the-difference-between-library-and-framework/ [Accessed: 24 Mar 2019.].

[20] Nishant Srivastava. Things i wish i knew when i started building android sdk/libraries, 2017. https://medium.com/p/dba1a524d619 [Accessed: 25 Mar 2019.].

[21] Mattt Thompson. Namespacing, 2014. https://nshipster.com/namespacing/ [Accessed: 14 Apr 2019.].

[22] John Vlissides, Ralph Johnson, Richard Helm, and Erich Gamma. Design patterns, 1994.

[23] Brandon Wozniewicz. The difference between a framework and a library, 2019. https://medium.freecodecamp.org/bd133054023f [Accessed: 24 Mar 2019.].